Report No. 401

ANALYSIS OF ALGORITHMS FOR FINDING ALL SPANNING TREES OF A GRAPH

by

Stephen Martin Chase

October 19, 1970

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 401

ANALYSIS OF ALGORITHMS FOR FINDING ALL SPANNING TREES OF A GRAPH*

by

Stephen Martin Chase

October 19, 1970

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the following grants: US NSF GJ 217, US NSF GJ 812, and project BUILD. The last stage of thesis rewriting was supported by IBM. This work was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, October 1970.

ANALYSIS OF ALGORITHMS FOR FINDING ALL SPANNING TREES OF A GRAPH

Stephen Martin Chase, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1970

Relatively little attention has been paid to the problem of measuring the efficiency of graph algorithms. The fact that the amount of work required by most graph algorithms varies greatly and unpredictably with the structure of the graph to which it is applied, makes this problem both practically important and theoretically difficult.

Two major goals were set at the outset of this investigation: first, to investigate and develop general approaches and specific techniques for analyzing the efficiency of graph algorithms, and second, to test and illustrate some of these approaches and techniques by using them for the analysis and comparison of specific algorithms.

With respect to the first goal, empirical and analytical methods are discussed. The use of empirical methods is greatly facilitated by a Graph Algorithm Software Package, GASP, which is an extension of PL/1 and has sets and graphs as additional data types.

With respect to the second goal, the problem of finding all the spanning trees of a graph was chosen. All published algorithms are analyzed and compared. A new algorithm is described, analytically compared to the previous algorithms, and found to be superior. For example, on the complete graph on n nodes, cost (new algorithm) = cost (A)/$(\sqrt{2}^{n-1})$, where A is the most efficient previous algorithm.

ACKNOWLEDGMENT

The author wishes to thank Professor Jurg Nievergelt for his extraordinary assistance, advice, and encouragement during the preparation of this thesis.

The support of the author's graduate education by the Department of Computer Science, University of Illinois at Urbana-Champaign, is gratefully acknowledged. The thesis research was supported by the following grants: US NSF GJ 217, US NSF GJ 812, and project BUILD. The last stage of thesis rewriting was supported by IBM.

The author greatly appreciates the typing by Mrs. Joanne Bennett. The efforts of the many people who aided the preparation of this thesis are also appreciated.

Finally, the author wishes to dedicate this thesis to his wife, Mary, and parents, Martin and Doris, for their sacrifices which made this effort possible.

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Goals of This Investigation

Graph theory and its applications have received much attention over the past two decades. In particular, many algorithms have been proposed for the solution of several graph problems which arise frequently in certain applications. However, relatively little attention has been paid to the problem of measuring the efficiency of these proposed algorithms. The fact that the amount of work required by most graph algorithms varies greatly and unpredictably with the structure of the graph to which it is applied, makes this problem both practically important and theoretically difficult.

Two major goals were set at the outset of this investigation: first, to investigate and develop general approaches and specific techniques for analyzing the efficiency of graph algorithms, and second, to test and illustrate some of these approaches and techniques by using them for the analysis and comparison of specific algorithms.

With respect to the first goal, there are two major categories of methods for measuring the efficiency of graph algorithms: empirical and analytical. Empirical methods are discussed in chapter 2, analytical methods in chapter 3. Empirical methods are greatly facilitated by a Graph Algorithm Software Package, GASP, which is described in detail in appendix 1.

With respect to the second goal, one graph problem, that of finding all the trees of a given graph, was chosen. It is discussed in chapter 4. There are many published algorithms for this problem, and these are described in chapter 5. By concentrating on these algorithms, useful techniques for efficiency analysis were developed and tested. Such detailed study also led to the development of a new algorithm for finding all the trees in a graph. This new algorithm is analytically compared to the best among the known algorithms in chapter 6 and is found to be superior to them.

1.2. Related Efforts

Previous efforts which share some of the goals of this investigation fall mainly into two categories: first, the analysis of specific algorithms, and second, the development of general purpose graph software. A brief review of some of the most relevant papers follow.

Authors of spanning tree algorithms sometimes present data on the performance of their algorithms ([Dawson 68], [Stehman 69]). This approach suffers from the fact that one cannot deduce the efficiency of an algorithm from its performance on isolated examples. A systematic comparison of seven algorithms on 13 graphs was done by Fernandez in his thesis [Fernandez 69a], and is mentioned in an abstract [Fernandez 69b].

Notable among the analyses of other graph algorithms are Gotlieb and Corneil's experiments with algorithms for finding a fundamental set of circuits ([Gotlieb 67], see also [Paton 69]), Shirey's analysis of algorithms for testing the planarity of graphs [Shirey 69], and Corneil and Gotlieb's analysis of an algorithm for testing graph isomorphism [Corneil 70].

The second category consists of papers which describe languages for graph processing ([Friedman, 69], [Hart, 69], [Read], [Wolfberg, 69]). These languages and GASP are similar in the sense that they all include graphs and sets as data types, and they all are extensions of an existing base language (e.g., FORTRAN, LISP). GASP is the only one which is an extension of PL/1, the richest widely available programming language.

1.3. Notation

Throughout this thesis, "n" will stand for the number of nodes in a given graph; "b" will stand for the number of branches; "t" will stand for the number of spanning trees; "c" will stand for the time cost of an algorithm.

Upper bounds on c will be expressed as $c = O(f(n,b,t))$, which means that there exists a constant A such that $c \leq A \cdot f(n,b,t)$ for sufficiently large n, b, and t. Similiarly, lower bounds will be expressed as $f(n,b,t) = O(c)$,

which implies that there exists a constant A such that $c \geq A \cdot f(n,b,t)$ for sufficiently large n, b, and t.

The cardinality of a set s will be denoted $|s|$. The symmetric difference (exclusive or) of sets s1 and s2 will be denoted s1 ⊕ s2.

Truth values, YES and NO, will be combined using "&", "or", and "¬".

A graph G consists of a set of words $\{v_1, v_2, \ldots, v_n\}$ and a set of branches $\{e_1, \ldots, e_b\}$. The set of branches incident to v_j will be denoted B_j. The degree of a node, "degree (v_j)", is equal to $|B_j|$.

2. EMPIRICAL METHODS OF MEASURING EFFICIENCY OF COMPUTATION

There are several methods which could be used to measure the efficiency of graph algorithms. These methods tend to fall into two categories, empirical and analytical. Of course, some methods have both empirical and analytical features, but the division into categories is still useful in order to understand general principles.

Empirical methods consist of implementing the algorithm on a computer, running several tests on it, measuring the cost, and drawing some form of conclusion from the observed data. This approach has several advantages and disadvantages.

2.1. Advantages

The first advantage is that empirical measures are often easier to obtain than analytic measures. This is especially true if one needs the implemented algorithm to solve problems. Obtaining data is relatively trivial. Interpreting the data may be easy (e.g., if one only wants to compare algorithms qualitatively to find out which algorithm is best), or may be very difficult (e.g., if one wants a quantitative prediction of the cost on graphs which have not been tested).

A second advantage occurs after a graph algorithm has been implemented: the programmer often sees ways to improve its efficiency (both on a programming level and a graph theoretical level). Insights into measuring the efficiency may occur as well.

Finally, empirical results produce numbers corresponding to actual run times which may prove to be more useful than analytically derived formulas which often yield only rates of growth.

2.2. Disadvantages

One major disadvantage of experimental testing of efficiency of graph algorithms is that the run time of an implemented program depends on many factors

which have little or nothing to do with the algorithm proper. These factors include the particular computer, language, and programmer; the implementation; and the method of representing graphs. A change in some of these factors could reverse the experimental conclusion of the superiority of one algorithm over another. Similarly, once the machine on which they were obtained becomes obsolete, experimental results are likely to lose their value.

The other major disadvantage of experimental testing is that because of computer time costs, only a small number of tests can be run. If the amount of computation required by the algorithm is sensitive to the structure of the graph, it becomes very difficult to accurately extend the results of tests on a small number of graphs to the class of all graphs.

Similarly, if the algorithm requires computation time which increases rapidly with increasingly large graphs, experimental measures will be limited to tests on small graphs. For tree-finding programs, 15-node graphs may be too large [Dawson 68]. Costs of algorithms applied to small graphs usually will permit only very poor extrapolations to the costs of larger graphs.

Many authors hide the inefficiency of their algorithms by illustrating them on small graphs where they appear reasonable. When applied to slightly larger graphs, the algorithms require considerably more computation.

2.3. Lessening the Disadvantages by Using Better Measuring Techniques

The two disadvantages mentioned in the previous section are due in varying degrees to the use of data consisting of computer run times. In order to obtain data dependent on properties of the algorithm rather than on the particular computer system used, the following technique can be used.

Divide the given program into logical groups of operations which have the property that during any test of the program, all the operations in a section will be executed the same number of times. Insert counters into the program, one to each logical section. Assign weights to each operation, and compute the total weight of a section as the sum of the weights of all the operations in the

section. Take the section counts from a computer test, multiply them by the corresponding weights, sum over all sections, and you have the total cost of that test graph.

This technique decreases the dependence on the particular computer system used because one can arbitrarily assign weights to operations in a manner consistent with any imaginable computer system. The cost then corresponds to the run time on the imaginary computer, perhaps quite different from the real time cost of the test run. Furthermore, many different costs (based on different imaginary machines) can be computed at the cost of just one real test. To achieve this, simply save the section counts and change the weight system.

This technique may increase slightly the size of test graphs which can be directly measured. Once the program has been debugged, any code which does not affect the flow of the program can be removed, reducing the real time required for a test without changing the computed cost.

GASP is very useful when the above technique is applied. GASP allows programmers to express graph and set operations in natural terms, without regard to how these objects are represented. Similarly, the operations on these objects are expressed independently of their implementation. Assigning a reasonable set of weights to GASP operations is easy.

Because programs written in GASP are independent of the representations, it is possible to run the same program with many different versions of GASP, thereby obtaining experience with different representations. GASP is structured so that small changes can be made in some GASP routines and data structures without requiring changes in the routines which use them.

3. ANALYTICAL METHODS OF MEASURING EFFICIENCY OF COMPUTATION

In contrast to empirical methods, analytical methods involve the mathematical analysis of the computational structure of algorithms. This approach also has its relative advantages and disadvantages.

3.1. Advantages

First, analytical results hold for arbitrarily large graphs, where experimental results would have to be extrapolated. Thus analytical results give a better indication of the true nature of the algorithm.

Second, analytical measures are usually performed on the algorithm proper rather than a machine-dependent implementation of the algorithm. Thus the results will not become obsolete when implementations improve.

3.2. Disadvantages

The big disadvantage with the analytical approach is that many graph algorithms are difficult to measure analytically, especially when the cost of the algorithm varies greatly with the structure of the graph (and not just its size). The goal of analytical methods is to express the cost in terms of a few easily calculated parameters of the input graphs. For some algorithms, this goal is unobtainable, and one must do at least one of the following:

1. Restrict the estimates and bounds to apply only to some subset of the set of all graphs.
2. Introduce more complicated parameters.
3. Accept larger measurement errors.

Another possible disadvantage of analytical measures is that they are derived for large graphs, so that small terms and details can be ignored. However, if for some reason the algorithm is applied only to small graphs, the ignored information may be more important than the derived formula.

3.3. Types of Analysis

There are several techniques which can be used in making analytical measures of efficiency. These techniques will be illustrated by applying them to an algorithm, A, of the following structure.

A: "Pick an arbitrary node X_0.
For all nodes X adjacent to X_0, do S."

S is an operation whose cost is large but constant, so that the total cost of A is determined by the number of executions of S.

Some of the techniques will be more significantly used (and therefore illustrated) in chapter 6.

A standard technique for measuring an algorithm's efficiency is worst-case analysis. If applied to algorithm A, the following analysis might take place. "The bound variable X takes its values from the set of nodes of the graph; therefore, n is a bound for the number of times S is executed. Hence, $c = O(n)$."

This method is usually the easiest to apply, but usually the least accurate. If an algorithm has $c = O(n^k)$ with k very small (say 2 or 3), then worst-case analysis may be accurate for some graphs. However, for less efficient algorithms or for typical graphs, the errors can grow rapidly and often become intolerable.

In order to get bounds which are tighter than those from worst-case analysis, it is usually necessary to make assumptions. That is, the test graphs are assumed to have certain properties. For example, assume all nodes have the same degree, d (which could be either a constant or some small function of n). Algorithm A would be analyzed as follows: "X will take on d values because that is exactly the number of nodes adjacent to X_0. Therefore, $c = O(d)$."

Assumptions should be chosen with care. Too many will make the analysis easy, but the conclusions will be of limited use. Too few may weaken the analysis so that only very loose bounds can be obtained.

Particularly useful assumptions are those which specify the test graphs in terms of one or more parameters. With such assumptions, analytic bounds can be derived and expressed in terms of the parameters.

For many algorithms, a useful one-parameter family of test graphs is the complete graph on n nodes. Complete graph analysis of Algorithm A would be as follows: "X_0 is adjacent to all of the other n-1 nodes; therefore, c = O(n-1)."

Other possible examples of parameterized classes of graphs include circuits of n nodes, ladders of r rungs, star graphs of b branches, rectangular grids of r rows and c columns, and others of even more parameters.

In addition to making the analysis easier, assumptions may be chosen in a way that reflects the intended use of the algorithm. For example, if the application is in electrical network theory, assumptions such as planarity or bounded degree of nodes may reflect physical limitations of the hardware.

The main disadvantage of these techniques is that the assumptions restrict the set of graphs for which the conclusions are valid. It is possible that the conclusions will be false for most graphs. This disadvantage is lessened when estimates which are derived on a small class of graphs can be used as bounds on a larger class. For example, if the cost of an algorithm increases whenever a non-parallel branch is added to the test graph, then the cost of that algorithm on the complete graph on n nodes will be an upper bound of the cost on any graph on n nodes.

When the task is to compare two or more algorithms and to determine which one is best, there are two approaches which can be used. The first approach is to apply the previously discussed techniques on each algorithm individually, and then compare the derived estimates and bounds. The second approach is to analyze directly the computational aspects of the differences between the competing algorithms.

To illustrate the second approach, suppose Algorithm B is obtained by

modifying Algorithm A so that X_0 is chosen to be a node of minimum degree. Then the comparison analysis may be as follows: "If the computation required in B to find a minimum degree X_0 is negligible, then Algorithm B is better than Algorithm A because S is executed fewer times."

One advantage of direct comparison is that the analysis is often easier, thus fewer (if any) assumptions will be required. With fewer assumptions, the conclusions will be valid for a larger set of graphs (perhaps all graphs).

Another advantage is that the inefficient parts of the algorithms are pinpointed. Such knowledge about the parts would be useful if it is possible to recombine the parts into new algorithms, or if analogous parts appear in another pair of algorithms.

A disadvantage of direct comparison is that numerical bounds for individual algorithms are not automatically produced. A related disadvantage is that this method cannot be used on an algorithm which has nothing in common with other algorithms.

4. THE PROBLEM OF FINDING ALL THE SPANNING TREES IN A GRAPH

4.1. The Problem: Its Variations and Applications

In order to make meaningful comparisons among graph algorithms, it is useful to focus on a single graph problem. For this thesis, the chosen problem is that of finding (i.e., listing exactly once) all the spanning trees of a connected undirected graph. A [spanning] tree is a set of [n-1] branches which are connected and contain no circuits.

There are several variations of the problem, including the following:

1. count the number of trees in any given graph [see section 5.2];
2. find formulas for the number of trees in special graphs ([Bercovici 69], [Cayley 89], [Mullin 67], [Myers 65], [O'Neil 66b], [Riordan 60]);
3. find all spanning trees of a directed graph ([Chen 66b, 67], [Paul 67]);
4. find all spanning trees common to two related graphs ([Ardon 69], [Mayeda 66, 68], [Stehman 69]);
5. find, for a given (directed or undirected) graph, all k-trees (which span k specified components), or all co-trees (complements of trees), or all sets satisfying certain conditions ([Berger 68], [Chen 65, 66a, 69a, 69d], [Dunn 68], [Mayeda 57], [Paul 67]);
6. find two spanning trees with minimal intersection [Chase];
7. find all rooted ordered trees of the complete graph [Scions 68].

Only spanning trees on undirected graphs will be considered for the remainder of this thesis, so the following conventions will be used. The term "tree" will mean spanning tree, "graph" will mean undirected graph, and "finding all trees" will mean listing all the trees of a given graph without duplications. Factoring the trees into (unions of) cartesian products is allowed; the applications (see below) can use answers in this form ([Bedrosian 62], [Chen 69a], [Dunn 68]).

The primary application of finding all trees is in the analysis of linear electrical networks ([Hakimi 66], [Stehman 69], [Weinberg 58]). A second application is in the analysis of multilevel masers [Bedrosian 62]. Other potential applications have been mentioned.

4.2. The Algorithms: Their Common Features

At least ten distinct algorithms for finding all trees have been proposed in the vast literature on this subject. In addition to their large number, these algorithms have other properties which make them highly desirable objects for efficiency measurements.

One property of these algorithms is that the cost, c (as well as the number of answers, t) grows exponentially with the size of the test graph. Exponential algorithms are desirable as objects of efficiency measurements (both empirical and analytical) because the large growth rates magnify the differences between algorithms. Thus the inferiority of a bad algorithm will be apparent even on small graphs. Competing exponential algorithms usually have a variety of growth rates, allowing analytical measurements to determine the most efficient algorithm, because only the growth rates of the costs of algorithms are considered in analytical measurements. Examples of competing algorithms which cannot be analytically contrasted because they share a common growth rate $[n^3]$ are the better algorithms for testing the planarity of graphs [Shirey 69].

Although different ideas for exponential algorithms can be contrasted by analytical measurements, differences in graph representation and differences in implementation efficiency do not show up. If an algorithm is more efficient in a particular representation, it will always pay to convert the input graph into that representation because the cost of conversion is $O(n^2)$ which will be small when added to an exponential term. Similarly, implementation improvements can do no better than to reduce the cost by a constant factor, which will not affect growth rates.

Another property of these algorithms is that analytical bounds are difficult

to derive (this explains the complete lack of meaningful bounds by the many authors of these algorithms). One of the reasons for this difficulty is that the cost of these algorithms depends greatly on the parameter t, which cannot be expressed in terms of n and b (except for a few special graphs). Fortunately, the scarcity of individual bounds in terms of n and b does not rule out comparison analysis. For example, any algorithm whose cost grows faster than t will be inferior to any algorithm whose cost grows slower than t.

A property of these algorithms which aids direct comparison analysis is that many of them can be arranged in a sequence in which the difference between one algorithm and the next is small. This property aids both the description and the analysis of the algorithms because only the differences need to be described and analyzed.

5. DESCRIPTION OF THE ALGORITHMS

This chapter briefly describes all known classes of algorithms for finding all trees in a graph. Sections 5.1 through 5.4 are independent of each other; section 5.5 describes a variation of the algorithm in section 5.4; section 5.6 introduces the remaining algorithms, each of which is described in terms of the differences from the preceding algorithm.

5.1. Exhaustion

Exhaustion algorithms simply search through a large set of candidate branch sets, testing each to see if it is a tree of the graph.

One algorithm ([Hale 61], [MacWilliams 58], [Mason 57], [Mayeda 57], [Weinberg 58]) generates all sets of $n-1$ branches from the graph, and tests each set to see if it is a tree.

Another algorithm ([Char 68], [Zobrist 64]) takes a previously computed list of all the trees on the complete graph on n nodes, and tests each tree to see if all of its branches belong to the input graph.

5.2. Determinants

The most efficient method to calculate t, the number of trees in a graph, is to evaluate a determinant [Harary 59]. Let M be a $n-1$ by $n-1$ matrix with entries m_{ij} defined as follows: m_{ii} = degree (v_i), and (for $i \neq j$) m_{ij} = - (the number of branches connecting v_i to v_j). Then, $t = \det(M)$ can be calculated by using any standard method of evaluating determinants (e.g., Gaussian Elimination).

Determinant algorithms ([Trent 54], [Weinberg 58]) to find all trees need to evaluate determinants symbolically, a complicated (and costly) process. Some of the more efficient "determinant" algorithms ([Chang 68]), [Chen 68], [Malik 67], [Nakagawa 58]) turn out to be different presentations of algorithms to be described later (sections 5.4, 5.7, and 5.10).

5.3. Decomposition

Many authors ([Berger 68], [Chen 69a, 69b], [Hakimi 64], [Jong 66], [Kim 60], [Lee 63], [MacWilliams 58], [Mayeda 59], [Myers 67], [Row 61], [Watanabe 61]) have suggested decomposition as a method to find all trees. The basic idea is to divide the graph into two or more subgraphs, to find the trees on these subgraphs, and then to combine these partial trees into trees of the input graph. Unlike most decomposition algorithms for other graph problems (e.g., planarity tests), the final step of combining the partial answers is not trivial.

There are other difficulties in constructing decomposition algorithms. Only a few algorithms ([Chen 69a, 69b], [Kim 60], [Mayeda 59], [Myers 67]) avoid duplication and its penalty of checking each tree against the list of trees. Some algorithms ([Chen 69a], [Lee 63], [MacWilliams 58], [Myers 67]) can be applied only to special types of graphs.

Apparently only one of these algorithms [Chen 69b] is general, avoids duplications, and overcomes some of the difficulties of combining partial trees. Like most of the references to decomposition algorithms, significant details are not specified, so no algorithm will be described (or analyzed) here. If the details could be worked out, a decomposition algorithm might be competitive with the best of the existing algorithms.

5.4. Tree Transformations

Several algorithms ([Chen 69c], [Fujisawa 59], [Hakimi 61, 66], [Kishi 69], [Malik 67], [Mayeda 65, 66, 68], [Stehman 69], [Watanabe 60], [Wing 63]) are based on "elementary tree transformations". Tree Y_1 is transformed by adding any new branch a_2 and removing any branch a_1 which lies in the path connecting the endpoints of a_2. The new tree $Y_2 = Y_1 \oplus \{a_1, a_2\}$.

For any two trees, Y_0 and Y, there is a sequence of trees $\langle Y_0, Y_1, Y_2, \ldots, Y_{k-1}, Y_k = Y \rangle$ such that for $1 \leq i \leq k$, Y_i is an elementary

tree transformation of Y_{i-1}. The "distance" from Y_0 to Y, denoted $d(Y_0,Y)$, is the minimum number of transformations necessary to change Y_0 into Y. For all Y and Y_0, $d(Y_0,Y) \leq n-1$.

A tree transformation algorithm begins with an initial tree Y_0. First, all possible elementary tree transformations are applied to Y_0 to get X_1, the set of all trees at distance 1 from Y_0. Next, X_2, the set of all trees at distance 2 from Y_0, is found by applying elementary transformations to the trees in X_1. Similarly, X_3 is found from X_2 by elementary transformations. This process continues until X_r is found where $r = \max[\text{over } Y]d(Y_0,Y)$.

The details of this algorithm, such as how to avoid duplications, will not be described here (see [Mayeda 65]).

On some graphs, the choice of Y_0 can make a big difference in the cost of the algorithm. The best Y_0 is a "central" tree ([Deo 66], [Malik 68]), for which $\max[\text{over } Y]d(Y_0,Y)$ is a minimum. One algorithm for finding a central tree has been suggested [Amoia 69].

5.5. Hamiltonian Paths

The trees of any graph can be arranged in a (Hamiltonian Path) sequence $Y_1, Y_2, \ldots, Y_t$ such that for $1 \leq i \leq t-1$, $d(Y_i,Y_{i+1}) = 1$ ([Chen 67], [Cummins 66], [Shank 68]). Algorithms to find trees in such an order have been suggested ([Kamae 67], [Kishi 67, 68]). These algorithms will not be described here because they are too complicated.

5.6. Introduction to Expansion Algorithms

The remaining algorithms (sections 5.7 through 5.12) expand the "variable Cartesian Product" $X_1 \times X_2 \times \ldots \times X_{n-1}$, where the definition of the set X_j depends on the choices of elements from X_1 through X_{j-1}. The basic flowchart for these algorithms appears in figure 1, and will be explained in detail in this section. Subsequent flowcharts will be described by explaining the changes in the contents of boxes 1 through 4.

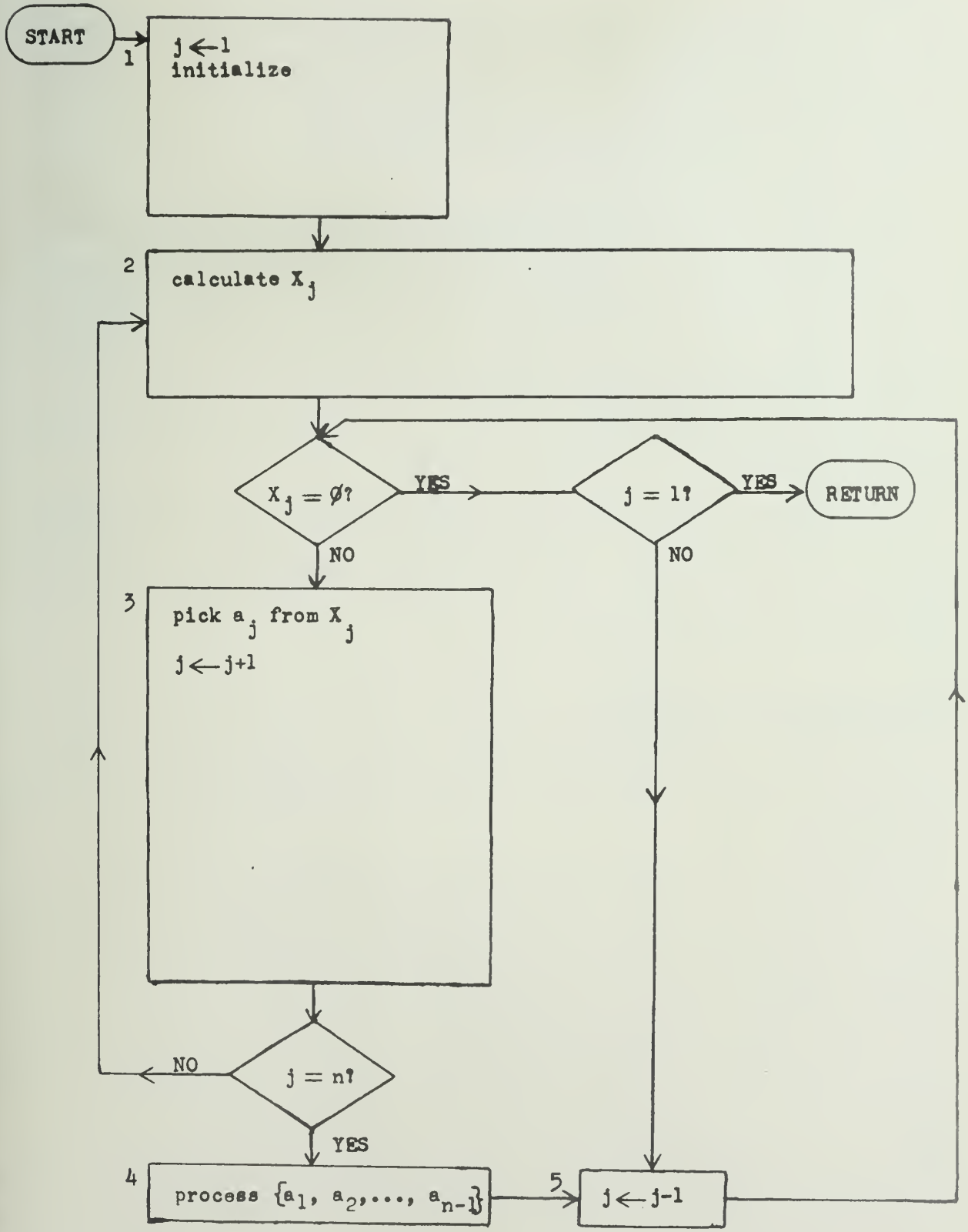

Figure 1: Expansion Algorithms

The two interlocking loops in the basic flowchart are roughly equivalent to $n-1$ nested loops (fixed nested loops cannot be used because n is a variable). The variable j specifies the nesting level. The "highest" level is 1, the "lowest" level is n.

Each level j $(1 \leq j \leq n-1)$, begins (box 2) with the calculation of X_j, a set of branches. X_j controls the iterations at level j. Namely, an iteration begins (box 3) with one branch being picked (and deleted) from X_j and being assigned to the bound variable a_j.

At the lowest level, the set $\{a_1, a_2, \ldots, a_{n-1}\}$ is processed as a tree candidate (box 4).

When computation at a level j is completed, the algorithm "backtracks" (box 5) to the previous level $j-1$ where another iteration (a_{j-1} from X_{j-1}) leads to a new instance of level j.

5.7. Cancellation of Non-Trees

This algorithm ([Bellert 62], [Chen 65], [Maxwell 66], [Piekarski 65]) is actually a method of expanding the symbolic determinant mentioned in section 5.2 [Myers 65].

The flowchart for this algorithm appears in figure 2. Box 2 reads "$X_j \leftarrow B_j - \{a_1, a_2, \ldots, a_{j-1}\}$" which means that X_j is the set B_j (all branches incident to node v_j) excluding any currently assigned a_i $(i = 1,2,\ldots,j-1)$. Box 4 reads "$L = L \oplus \{\{a_1, a_2, \ldots, a_{n-1}\}\}$", where L is a list of tree candidates which have been generated at previous instances of the lowest level. If $\{a_1, a_2, \ldots, a_{n-1}\}$ is equal to a set S already in L, then S is removed from L. $\{a_1, a_2, \ldots, a_{n-1}\}$ is added to L if and only if there is no such match. When the algorithm terminates, L is the list of all trees of the graph.

5.8. Circuit-Free Expansion

This algorithm ([Brownell 68], [Char 68], [Hobbs 59], [Mason 57]) differs

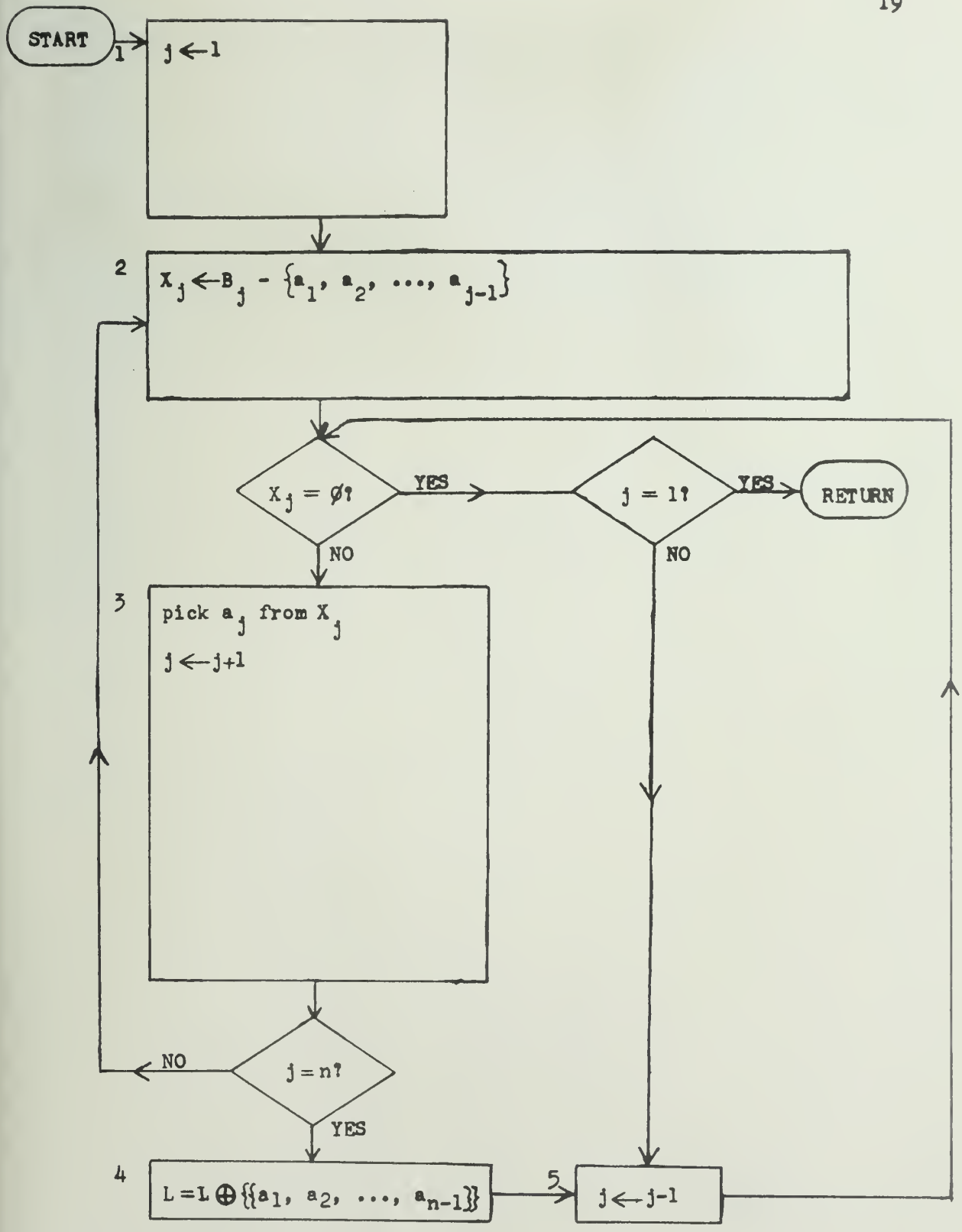

Figure 2: Cancellation of Non-Trees

from the previous algorithm by avoiding the generation of non-trees (rather than waiting to cancel them out of the list L). This algorithm rejects the choice of any branch which forms a circuit with previously chosen branches. At the lowest level, the tree candidate can be output immediately because any set of n-1 branches which does not contain a circuit is a tree.

The flowchart for this algorithm appears in figure 3. Box 2 now reads "$X_j \leftarrow B_j$ - Circuit_Makers $(a_1, a_2, \ldots, a_{j-1})$". Box 4 now reads "Output $\{a_1, a_2, \ldots, a_{n-1}\}$".

5.9. Connected Expansion

This algorithm ([Berger 67], [Cummins 64], [Feussner 02, 04], [Hirayama 63], [Minty 65], [O'Neil 66a]) differs from the previous algorithm in the method of avoiding non-trees. Instead of testing for circuits, this algorithm preserves connectedness. The references cited above offer a variety of algorithms; an efficient representative is described below.

The flowchart for this algorithm appears in figure 4. The new variables are Y_j (needed to avoid duplications) and p_j (representing the nodes in the current connected subgraph). Box 1 has added the initializing statements "$Y_1 \leftarrow$ Branches of Graph" and "$p_1 \leftarrow v_1$". Box 2 now reads "$X_j \leftarrow$ Boundary $\{p_1, p_2, \ldots, p_j\} \cap Y_j$" which means that X_j contains all branches (in Y_j) which have exactly one endpoint belonging to $\{p_1, p_2, \ldots, p_j\}$. This guarantees that any branch picked from X_j will preserve connectedness and avoid circuits. Box 3 has added the statement "$p_{j+1} \leftarrow$ other_endpoint (a_j)", which means that the endpoint of branch a_j which is not already in $\{p_1, p_2, \ldots, p_j\}$ is assigned to the node variable p_{j+1}. Also in box 3 are the statements "remove a_j from Y_j" and "$Y_{j+1} \leftarrow Y_j$", which limit the choice of branches at lower levels (see box 2) in order to avoid duplications.

5.10. Factoring

This algorithm ([Ardon 69], [Chang 68], [Chen 68, 69c], [Cummins 64],

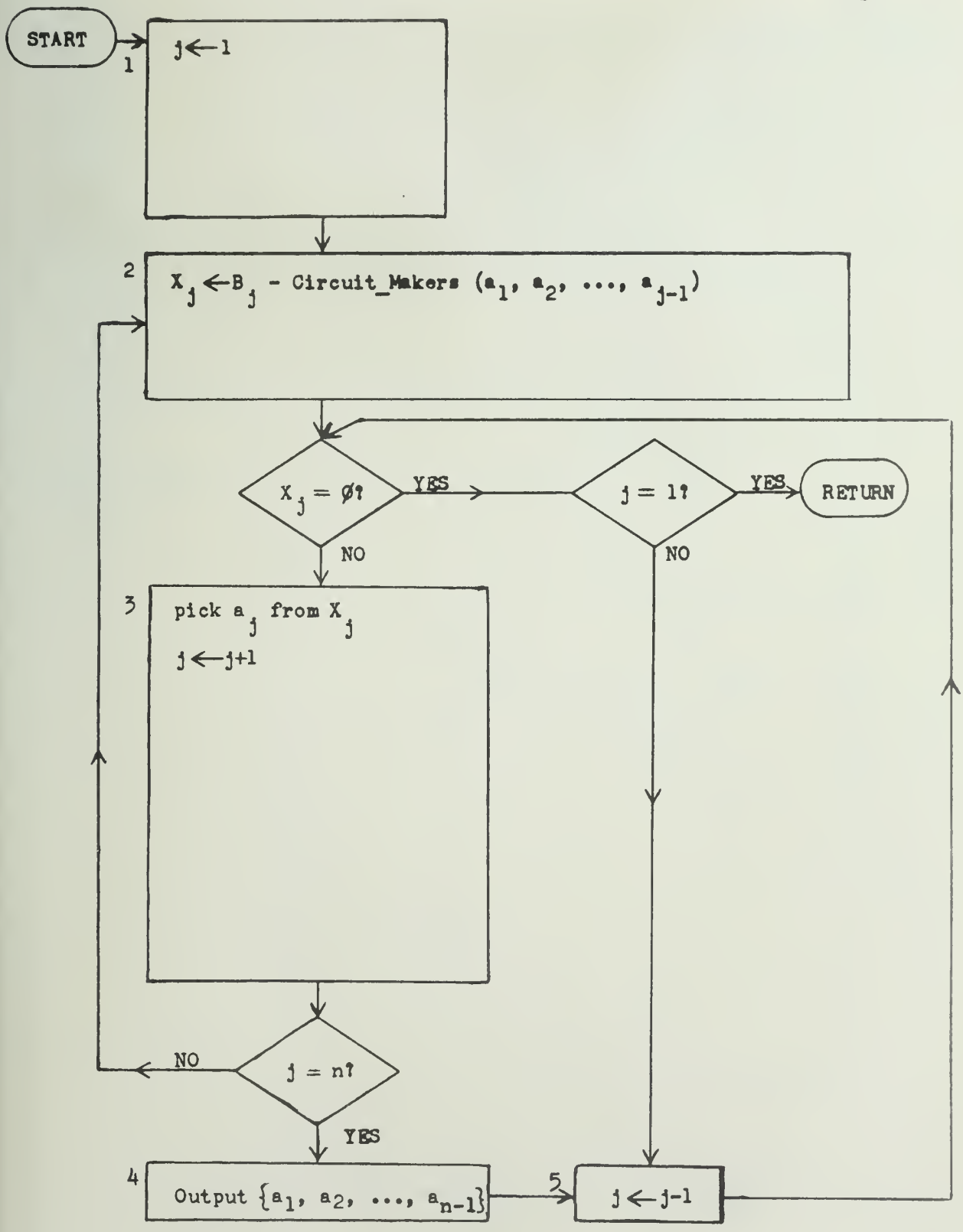

Figure 3: Circuit-Free Expansion

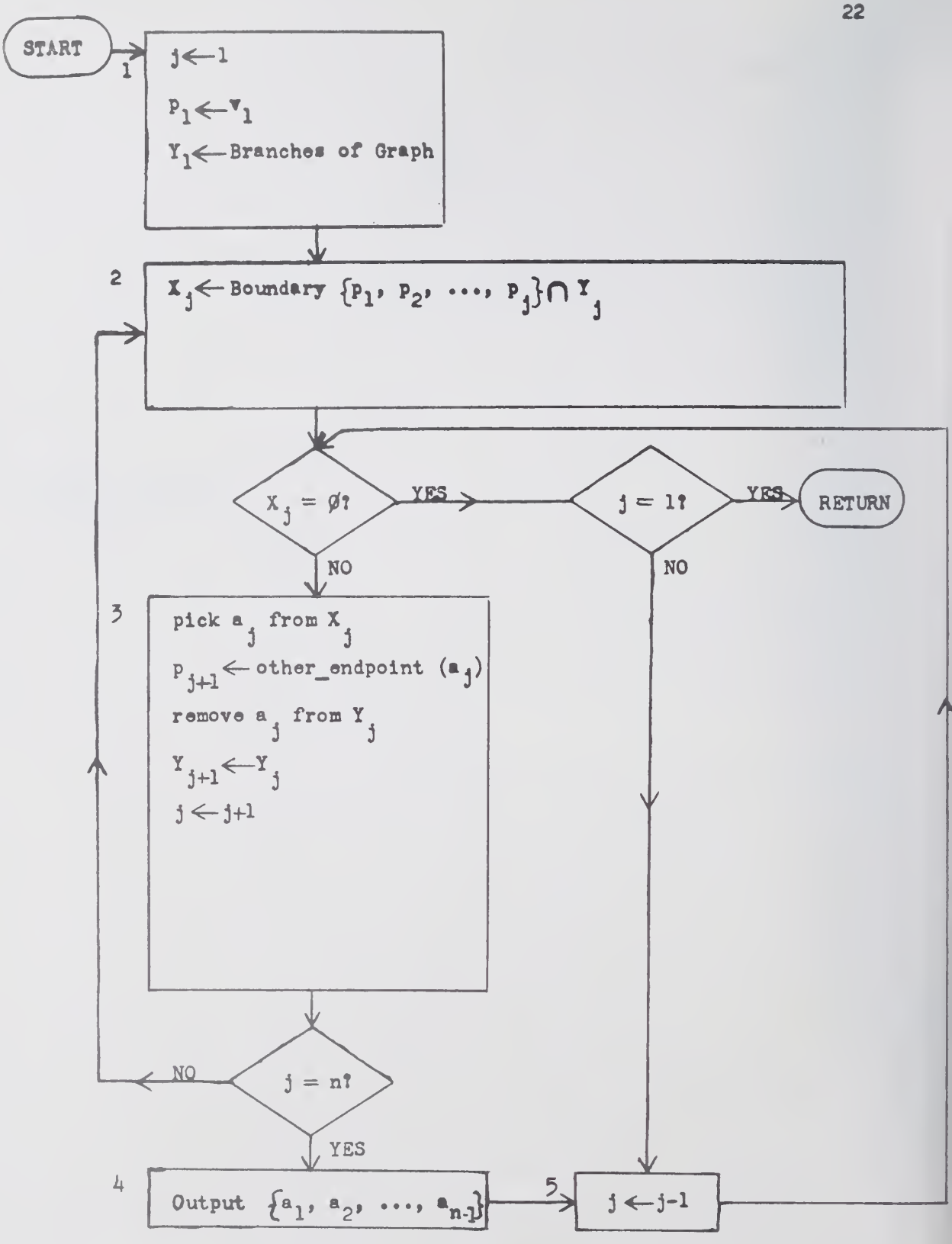

Figure 4: Connected Expansion

[Hirayama 65], [Holt 68], [Mason 57], [McIlroy 69], [Nakagawa 58], [Percival 53]) differs from the previous algorithm in that when node p_j is added to the currently connected subgraph, all branches in X_j which are incident to p_j are factored together into a single iteration. As a consequence, at the lowest levels, instead of individual trees of n-1 branches, Cartesian products of n-1 factors are produced.

The flowchart for this algorithm appears in figure 5. In box 3, "pick A_j from X_j" and "$p_{j+1} \leftarrow$ other_endpoint (A_j)" mean that a_j is picked from X_j and p_{j+1} is the other endpoint of a_j (as in the previous algorithm). However, a_j is now extended to A_j, a factor set of branches: $A_j = \{a_j\} \cup X_j \cap B_{p_{j+1}}$; that is, A_j contains all branches in X_j which are incident to p_{j+1}. All of A_j is removed from X_j. Similarly, "remove A_j from Y_j" deletes the entire subset A_j from Y_j.

In box 4, "output $A_1 \times A_2 \times \ldots \times A_{n-1}$" means that a family of trees is output in the form of a Cartesian product of the factor sets A_j, $1 \le j \le n-1$. This factored form is adequate for the applications (see section 4.2), but if individual trees are desired, they can be obtained by finding all combinations of one branch from each of the n-1 factor sets (this Cartesian product expansion could be accomplished by the flowchart in figure 1, with box 2: "$X_j \leftarrow A_j$" and box 4: "Output $\{a_1, a_2, \ldots, a_{n-1}\}$").

5.11. More Factoring

The idea behind this algorithm is to factor into a single iteration (the last one) all those cases in which only one branch from X_j appears in a tree. To avoid duplication, the other (earlier) iterations from X_j lead to the choice (at level j+1) of an additional branch of X_j.

The flowchart for this algorithm appears in figure 6. The new variables are d_j (a truth value which controls X_j) and Z_j (temporary storage for X_j). Box 2 now reads "if d_j then $Z_j \leftarrow X_j \leftarrow$ Boundary $\{p_1, p_2, \ldots, p_j\} \cap Y_j$

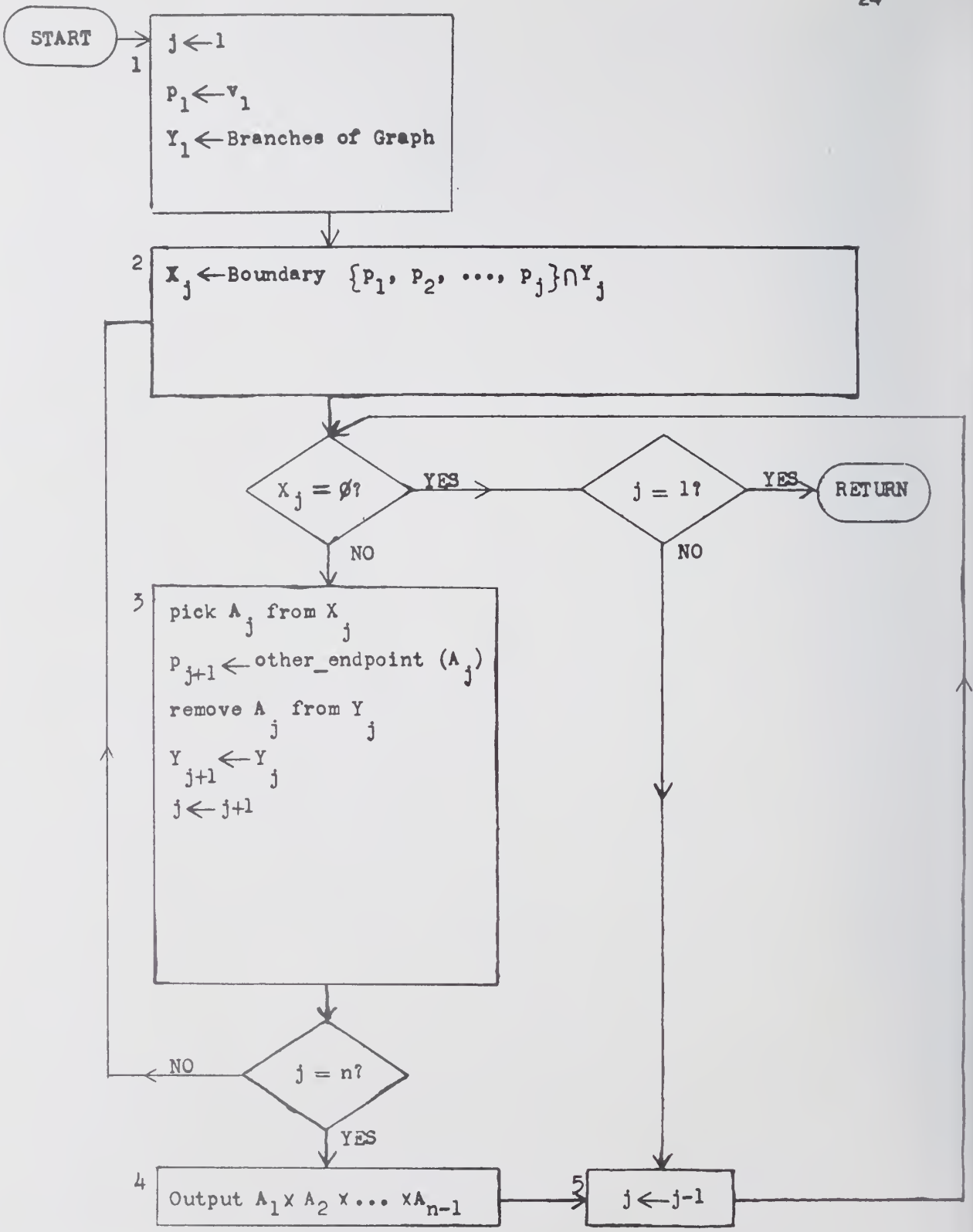

Figure 5: Factoring

else $X_j \leftarrow X_{j-1}$". This means that if d_j = YES, then X_j is calculated as in the previous algorithm, and stored in Z_j. If d_j = NO, then X_j is assigned the current value of X_{j-1} (since one branch was picked from X_{j-1} at level j-1, this guarantees another branch from X_{j-1} at level j).

Box 3 has the additional statements "$d_{j+1} \leftarrow (\neg d_j)$ or $(X_j = \phi)$" and "if d_j & d_{j+1} then $A_j \leftarrow Z_j$". Thus, if d_j = NO then $d_{j+1} \leftarrow$ YES. If d_j = YES and X_j is not empty, then $d_{j+1} \leftarrow$ NO. If d_j = YES and X_j is empty, then $d_{j+1} \leftarrow$ YES and A_j is replaced by Z_j (the saved value of the full set X_j before deletions).

5.12. Pruning

In the algorithms of sections 5.9 through 5.11, branches are deleted from the Y_j's. Even though the input graph was connected, deleted branches may cause $YA_j = Y_j \cup (A_1 \times A_2 \times \ldots \times A_{n-1})$ to fail to connect all the nodes (denote this situation by "YA_j fails"). Once YA_j fails, further computation at levels j through n-1 is wasted because no spanning trees can be found on a disconnected graph. Thus it would be useful to know when YA_j fails. On the other hand, an additional connectedness test would be expensive because it would be executed so many times.

The algorithm of this section differs from the three previous algorithms in that needless computation is avoided when YA_j fails, but an additional test for connectedness is not needed. This is accomplished by using "failure to find trees" as a test for connectedness.

The flowchart for this algorithm appears in figure 7. The new variable is k_j (the count of executed iterations from X_j). Box 2 initializes this count: "$k_j \leftarrow 0$". Box 3 increments the count "$k_j \leftarrow k_j + 1$".

The major change occurs in the NO branch of the "j=1?" decision box where a further test is inserted: "$k_j = 0$?", which means "was X_j empty in box 2?". If the answer is NO , then computation proceeds (as in previous

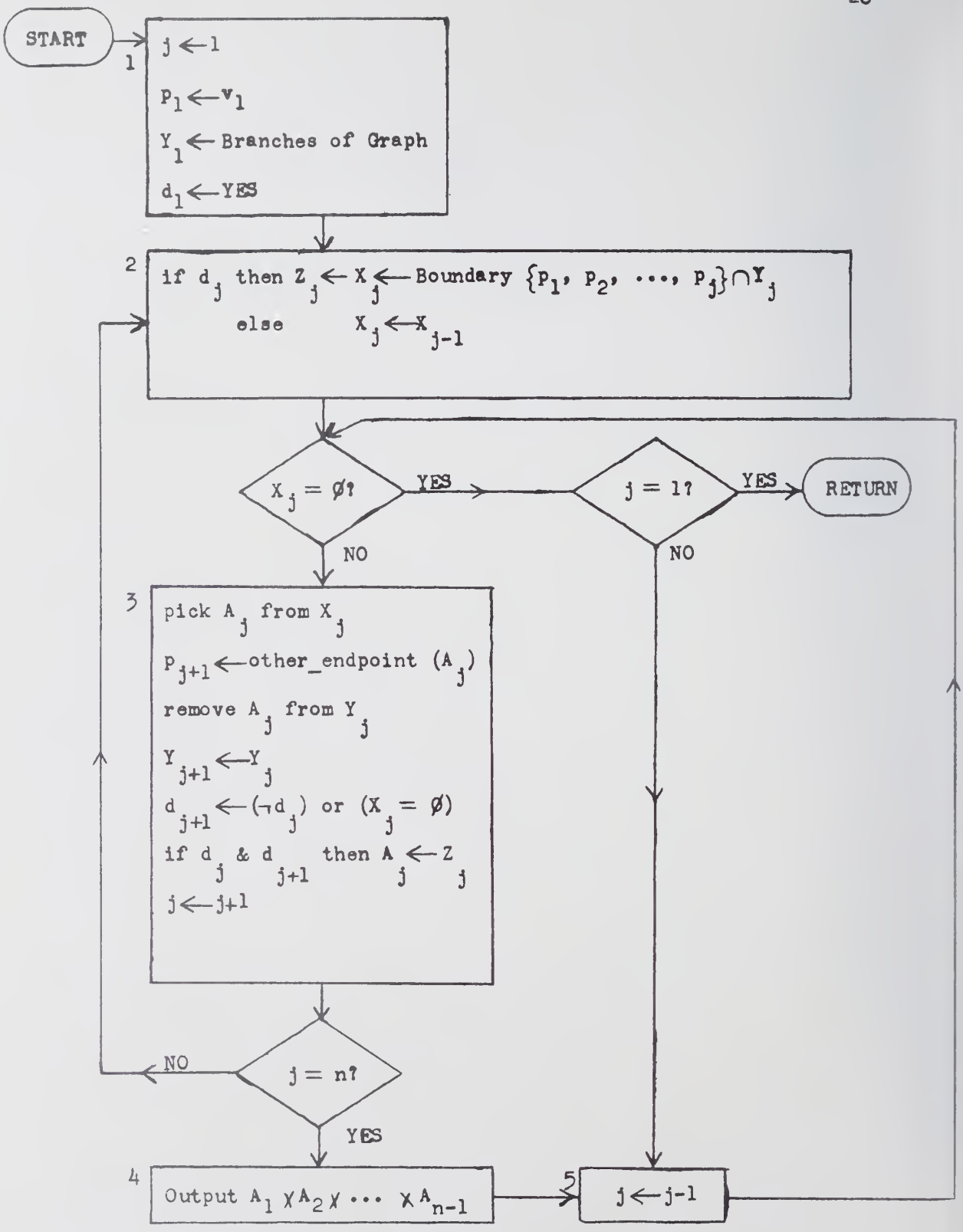

Figure 6: More Factoring

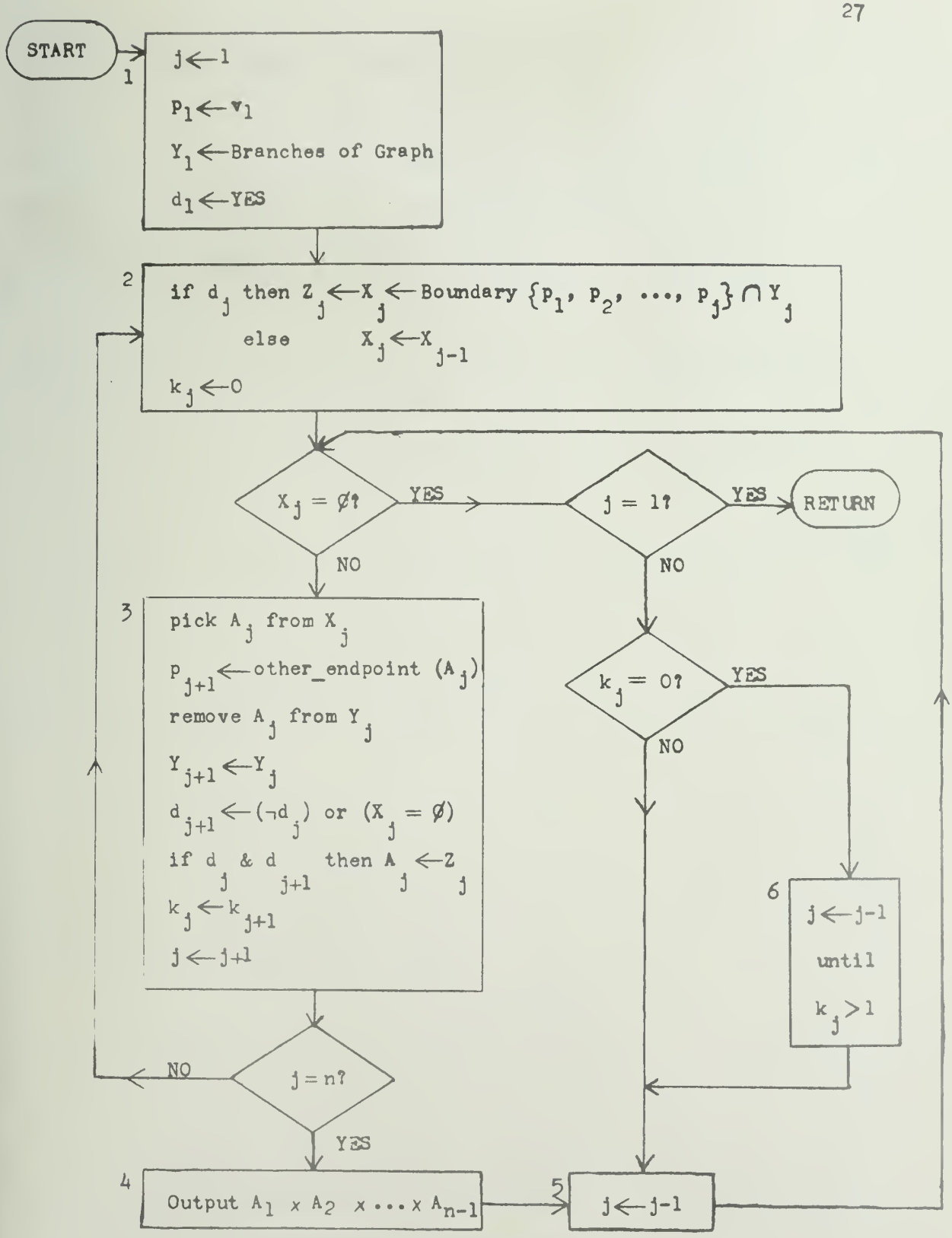

Figure 7: Pruning [the New Algorithm]

flowcharts) to box 5. If the answer is YES, then it is known that YA_j fails, and computation proceeds to box 6 which reads "$j \leftarrow j-1$ until $\ell_j > 1$". This means that control returns to the previous level ($j \leftarrow j-1$) and continues to return to higher levels (pruning unnecessary iterations) until $\ell_j > 1$. The remaining iterations at level j are then pruned by proceeding to box 5 ($j \leftarrow j-1$).

The idea behind box 6 is that the first iteration of box 3 does not change the value of YA_j from that of YA_{j-1}. Thus the final value of j leaving box 6 indicates the highest level at which YA_j failed.

5.13. Variations

There are many variations of the algorithms of sections 5.7 through 5.12. Some will be mentioned very briefly in this section.

There is an algorithm, Circuit Check, which is "half way" between Cancellation of Non-Trees [5.7] and Circuit-Free Expansion [5.8]. This algorithm is useful for analysis and will be described in section 6.4.

The algorithms of sections 5.7 and 5.8 can be generalized [Maxwell 66] by replacing B_j with a somewhat more general cutset.

Sections 5.7 and 5.8 can be improved by labeling the nodes so that degree $(v_j) \leq$ degree (v_{j+1}). For sections 5.9 through 5.12, a good heuristic is to always choose p_j to be of minimal degree.

For graphs with $b < 2(n-1)$, it may pay to find all co-trees (using some form of duality) and convert them to trees.

Finally there are many special cases which can occur in graphs (either initially or during computation) which can be handled more efficiently than the general case. For example, the existence of separating nodes or separating branches allows a quick decomposition.

6. ANALYTICAL MEASUREMENTS OF SELECTED ALGORITHMS

The algorithms described in chapter 5 will now be measured by the techniques described in section 3.3. A priori bounds (which do not depend on the structure of an algorithm) are given in section 6.1. An example of worst case analysis appears in section 6.2. In the remaining sections, only the expansion algorithms [5.6 through 5.12] are analyzed, using the "computation tree" defined in section 6.3. Section 6.4 employs direct comparisons of consecutive algorithms. Section 6.5 introduces the "quotient operator" and uses it to measure the New algorithm on "closed ladder" graphs. Finally, section 6.6 applies "complete graph analysis" in order to obtain upper bounds for the factoring algorithms.

6.1. A Priori Bounds

Sometimes it is possible to derive bounds for an algorithm without knowing its structure. If an algorithm is difficult to analyze, a priori bounds may be the tightest available bounds.

Find-all-trees algorithms illustrate this point because the required number of answers, t, grows exponentially. Any algorithm which finds trees one at a time must have $t = O(c)$ [recall from section 1.3 that "$f(n,b,t) = O(c)$" means $\exists A$ such that $c \geq A \cdot f(n,b,t)$]. For the algorithms of sections 5.4, 5.5, 5.8, and 5.9, tighter bounds are difficult to obtain.

Another example of "a priori" bounding occurs in the exhaustion algorithms (section 5.1). The first algorithm checks all $\binom{b}{n-1}$ combinations of $n-1$ branches, so regardless of the details, $\frac{b!}{(b-n+1)!(n-1)!} = O(c)$. The second algorithm checks each of the n^{n-2} [Cayley 89] trees of the complete graph, so $n^{n-2} = O(c)$. The storage required by the second algorithm is also larger than n^{n-2}. These lower bounds are sufficient to demonstrate the inefficiency of these algorithms.

6.2. Worst Case

This section will illustrate "worst case analysis" as applied to the first exhaustion algorithm [5.1]. Let the branches of the graph be numbered 1 through b. Represent each combination of branches by an ordered list of n-1 positions, p_j, each position containing a branch number ($p_j = i$, where $1 \le i \le b$). In order to avoid duplications, require that $p_j < p_k$ for $j<k$. In order for this condition to be satisfied for all j and k, each p_j must be restricted to values from a set of b-n+2 branch numbers, namely $j \le p_j \le b-n+1+j$ $(1 \le j \le n-1)$.

Assume that the algorithm is equivalent to n-1 nested iteration loops, each loop corresponding to a position in the ordered list. Since each loop goes through a maximum of b-n+2 iterations, the code in the innermost loop is executed no more than $(b-n+2)^{n-1}$ times. If that code (a tree test) costs O(n), then $c = O(n \cdot (b-n+2)^{n-1})$.

6.3. Computation Trees

A computation tree, CT(G, A), traces the execution of algorithm A applied to graph G. The expansion algorithms have computation trees of height n, as shown in figure 8. The meaning of CT(G, A) and the definition of its size parameters $c_i(G, A)$, are as follows.

The nodes of CT(G, A) are arranged in n levels, directly corresponding to the n levels of the flowchart of A. At the bottom (i.e., level n) of CT(G, A), there are $c_4(G, A)$ nodes, each corresponding to an execution of box 4 ("process $\{a_1, a_2, \ldots, a_{n-1}\}$") of the flowchart of A. Each of the other $c_2(G, A)$ nodes corresponds to an execution of box 2 ("compute X_j"). There are $c_3(G, A)$ branches in CT(G, A), each connecting a node on level j to one on level j+1 (corresponding to an execution of box 3: "pick a_j from X_j"). Since the number of nodes equals the one plus number of branches, $c_2(G, A) + c_4(G, A) = 1 + c_3(G, A)$. The arguments G and A may be modified or dropped when no confusion would result.

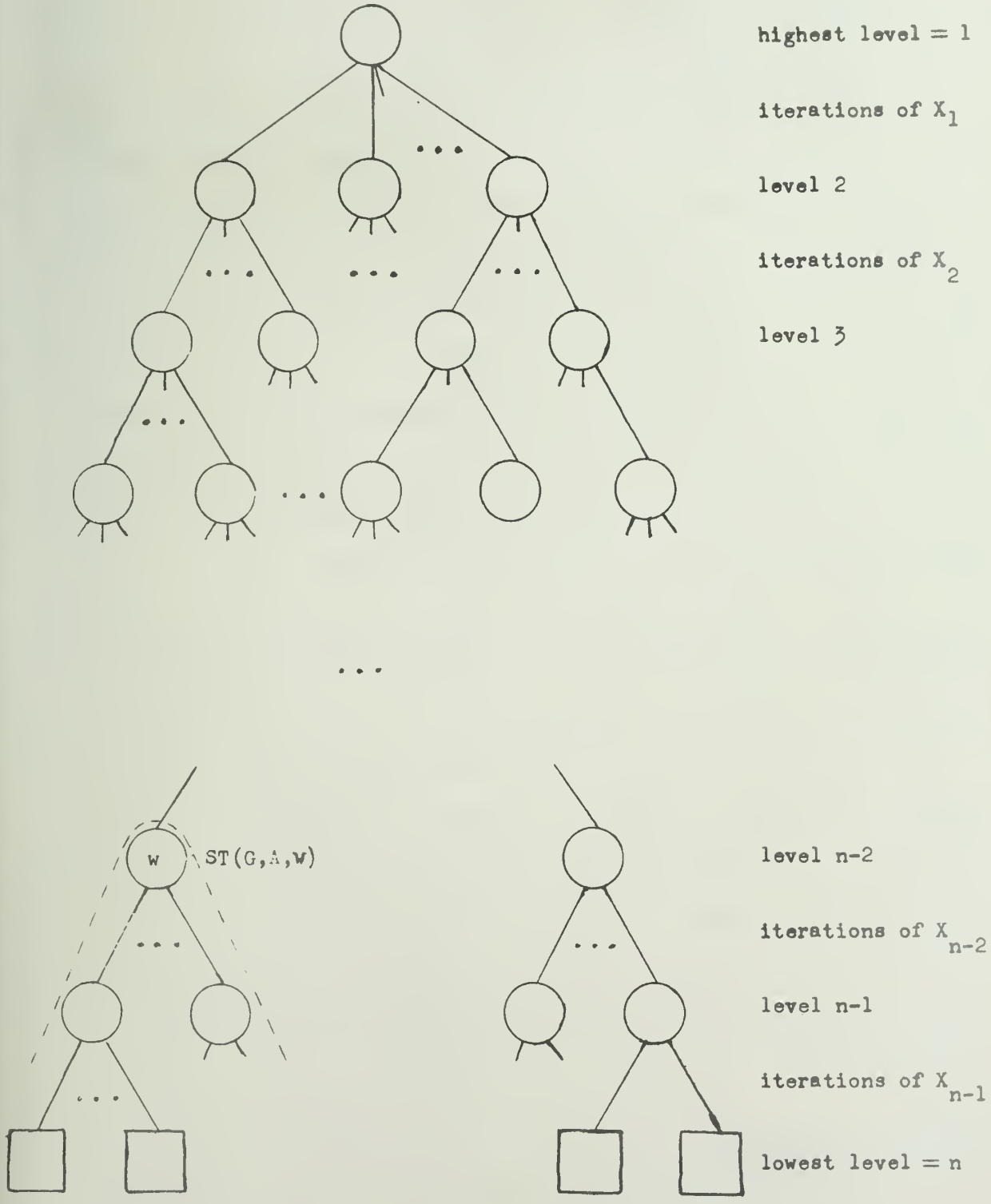

Figure 8: Computation Tree CT(G,A)

The cost of algorithm A applied to G can be expressed in terms of these size parameters. Using the last equality and ignoring very small costs, $c = c_2(G, A)[\text{cost(box 2)} + \text{cost(box 3)}] + c_4(G, A)[\text{cost(box 4)} + \text{cost(box 3)}]$.

Using this formula, a worst case bound for the Cancellation of Non Trees (CNT) algorithm of section 5.7 can be derived as follows. First, cost (box 4) = cost (compare two sets) · (# of sets in L). Intuitively and in practice, (# of sets in L) = O(t). Since cost (compare) = O(n), cost (box 4) = O(nt). Since this dominates the costs of the other boxes, only $c_4(G, CNT)$ needs to be calculated. Now, from the flowchart (figure 2), $X_j \leq B_j$. Therefore, in CT(G, CNT) each node at level j leads to (at most) degree (v_j) nodes at level j+1. Thus $c_4(G, CNT) \leq \prod_{j=1}^{n-1}$ degree (v_j). To express c_4 in terms of n and b, first use the "geometric vs. arithmetic mean" inequality [i.e., $\sqrt[n]{x_1 \cdot x_2 \ldots x_n} \leq \frac{1}{n}(x_1 + x_2 + \ldots + x_n)$], to obtain $c_4(G,CNT) \leq (\frac{1}{n-1} \sum_{j=1}^{n-1}$ degree $(v_j))^{n-1}$. Assuming degree $(v_n) \geq$ degree (v_j) [the most efficient case], $c_4(G,CNT) \leq (\frac{2b}{n})^{n-1}$. Thus $c = O(nt(\frac{2b}{n})^{n-1})$.

In the analysis to follow, it will be convenient to use the notation ST(G, A, w) to denote the subtree of CT(G, A) consisting of the node w [$w \in$ CT(G, A)] and all the nodes and branches connected to w from below. If w_1 is the root node, $ST(G, A, w_1) = CT(G, A)$. Two subtrees, $ST(G_1, A_1, w_1)$ and $ST(G_2, A_2, w_2)$, are isomorphic if there is a one-to-one and onto mapping of the nodes and branches which preserves incidence and level relationships.

6.4. Direct Comparisons

The expansion algorithms (5.7 through 5.12) will now be sequentially compared in terms of their computation trees. The expression "$c_i(A_1) \leq c_i(A_2)$" means that for all graphs G, and for each size parameter c_i (i = 2, 4), $c_i(G, A_1) \leq c_i(G, A_2)$.

The Circuit-Free Expansion (CFE) algorithm [5.8] introduces two changes from the previous algorithm (CNT). First, the "non-tree test" changes from "check L for duplicates" to "check for circuits". Second, this test has been moved up from box 4 to box 2. In order to isolate the change in efficiency, let us make these changes one at a time.

If the second change is made without the first [Piekarski 65], the cost increases [based on limited empirical evidence]; therefore, let us try the first change without the second. Call this the Circuit Check algorithm (CtC). Since the only change is in box 4 ("check $\{a_1, a_2, \ldots, a_{n-1}\}$ for circuits"), the computation tree does not change: $c_i(\text{CtC}) = c_i(\text{CNT})$. However, cost (box 4) drops from $O(n \cdot t)$ to $O(n)$ [the cost of a circuit test]. Clearly, Circuit Check is more efficient than Cancellation of Non-Trees.

Now add the second change. The cost of each box is $O(n)$ regardless of the placement of the circuit test (only the constants change). However $c_i(\text{CFE}) \leq c_i(\text{CtC})$ because the non-trees are discovered sooner, and needless computation is avoided. For nearly all graphs G, $c_i(G, \text{CFE}) < c_i(G, \text{CtC})$. Thus the second change is an improvement also.

The Connected Expansion (Con) algorithm [5.9] will be considered equally efficient as Circuit-Free Expansion. Both algorithms find trees one at a time, avoiding non-trees and duplications, so $c_4(\text{CFE}) = c_4(\text{Con}) = t$. Empirically, Circuit-Free Expansion appears more efficient [Fernandez 69a].

The Factoring (Fac) algorithm [5.10] is clearly an improvement over "one tree at a time" algorithms. Each factor $A_j = \{a_j^1, a_j^2, \ldots, a_j^k\}$ (box 3, figure 5), corresponds to a node w_{j+1} in CT(G, Fac); i.e., A_j corresponds to the entire subtree ST(G, Fac, w_{j+1}). Each a_j^i corresponds to a node w_{j+1}^i in CT(G, Con). Therefore, ST(G, Fac, w_{j+1}) replaces k subtrees ST(G, Con, w_{j+1}^i), $i = 1, \ldots, k$. Clearly $c_i(G, \text{Fac}) \leq c_i(G, \text{Con})$, with equality holding only if G is a tree. Typically, $c_i(G, \text{Fac})/t$ (the "cost per

tree") goes to zero exponentially as n increases [6.6, figure 10].

For each X_j calculated in box 2 of the Factoring algorithm, the trees which contain just one branch from X_j will be calculated k times over, where k is the number of iterations necessary to empty X_j. The More Factoring (MF) algorithm [5.11] combines these k cases into a single iteration, clearly an improvement in efficiency. Thus $c_i(G, MF) \leq c_i(G, Fac)$, with equality holding rarely. Typically, $c_i(MF)/c_i(Fac)$ goes to zero exponentially as n increases [6.6]. An intuitive indication of the improvement is that the factors are larger; i.e., on a complete graph, Factoring will always find at least one Cartesian product family consisting of a single tree [Chang 68], while (if $n>3$) every family found by More Factoring will contain at least two trees.

The Pruning algorithm [5.12] is clearly an improvement. The test for correctedness is obtained at negligible cost, but the potential savings are large. Naming this algorithm (with factoring and pruning, [figure 7]) the New algorithm, $c_i(G, New) \leq c_i(G, Fac)$.

6.5. Special Graphs: The Quotient Operator

This section (as well as the next) will illustrate the technique of measuring the cost of an algorithm on a parameterized class of graphs, $G = G(p)$. For the algorithms to be measured, $c = k_2\, c_2(G) + k_4\, c_4(G)$ with $k_i = O(n)$; thus, the only quantities which need to be measured are $c_2(G)$ and $c_4(G)$. Since $G = G(p)$, $c_i(p)$ will replace $c_i(G)$. Only algorithms with factoring will be measured directly; the "one at a time" algorithms [5.4, 5.8, 5.9] have $c_4(p) = t(p)$, the number of trees as a function of p.

For the classes of graphs to be considered, $c_2(p)$, $c_4(p)$, and $t(p)$ are all exponential in p. In order to derive, compare, and plot these functions, $f(p)$, the "quotient operator", $Q(f, p)$, will be used: $Q(f, 1) = f(1)$, and for $p>1$, $Q(f, p) = f(p)/f(p-1)$ [it is not difficult to interpret

$f(1) \geq 1$, thus $Q(f, 2)$ is well defined]. Clearly, $f(p) = \prod_{i=1}^{p} Q(f, i)$. For the functions to be considered, there will always exist a "quotient limit", $q(f, p)$, either a constant or a linear function of p, such that $\lim_{p \to \infty} \frac{Q(f, p)}{q(f, p)} = 1$. For example, $q(p!, p) = p$, $q(k^p, p) = k$, $q(k^p, k) = 0$.

As a first example of a special class of graphs, consider $L(r)$, the closed ladder of r rungs (see figure 9). Since $n(r) = 2r$ and $b(r) = 3r-2$, this example will show that even on graphs with rank > nullity [i.e., $b < 2(n-1)$], the New algorithm has cost per tree, c/t, going to zero exponentially. To prove this claim, it suffices to show that $q(c_i(r, \text{New}), r)/q(t, r) < 1$.

It is not difficult to derive the equation $t(r) = 4\,t(r-1) - t(r-2)$, with $t(1) = 1$, $t(2) = 4$. In fact, $t(r) = t(i+1) \cdot t(r-i) - t(i) \cdot t(r-i-1)$, for any i, $1 \leq i \leq r-2$. From this equation it is not difficult to show that $q(t, r) = 2 + \sqrt{3}$.

The New algorithm on $L(r)$ starts at p_1, one of the four corners (nodes of degree 2). There will be two iterations at the first level of the computation tree $CT(r)$ [see figure 9].

The first iteration handles the case in which both branches incident to p_1, the starting node, are included in a tree. Thus at level 3, the three branches in X_3 [see figure 9] will reach only two nodes. Because of factoring, the remaining computation from X_3 is just like starting at a corner of $L(r-1)$. If w_3 is the node in $CT(r)$ which corresponds to the calculation of X_3, then $ST(r, w_3)$ is isomorphic to $CT(r-1)$.

The second iteration at level 1 handles the case in which only one of the branches incident to p_1 will be included in a tree. Assuming p_2 is chosen to be the corner adjacent to p_1, then there is only one branch in X_2, and this leads to a corner of $L(r-1)$ [see figure 9]. Once again, if w_3' is the

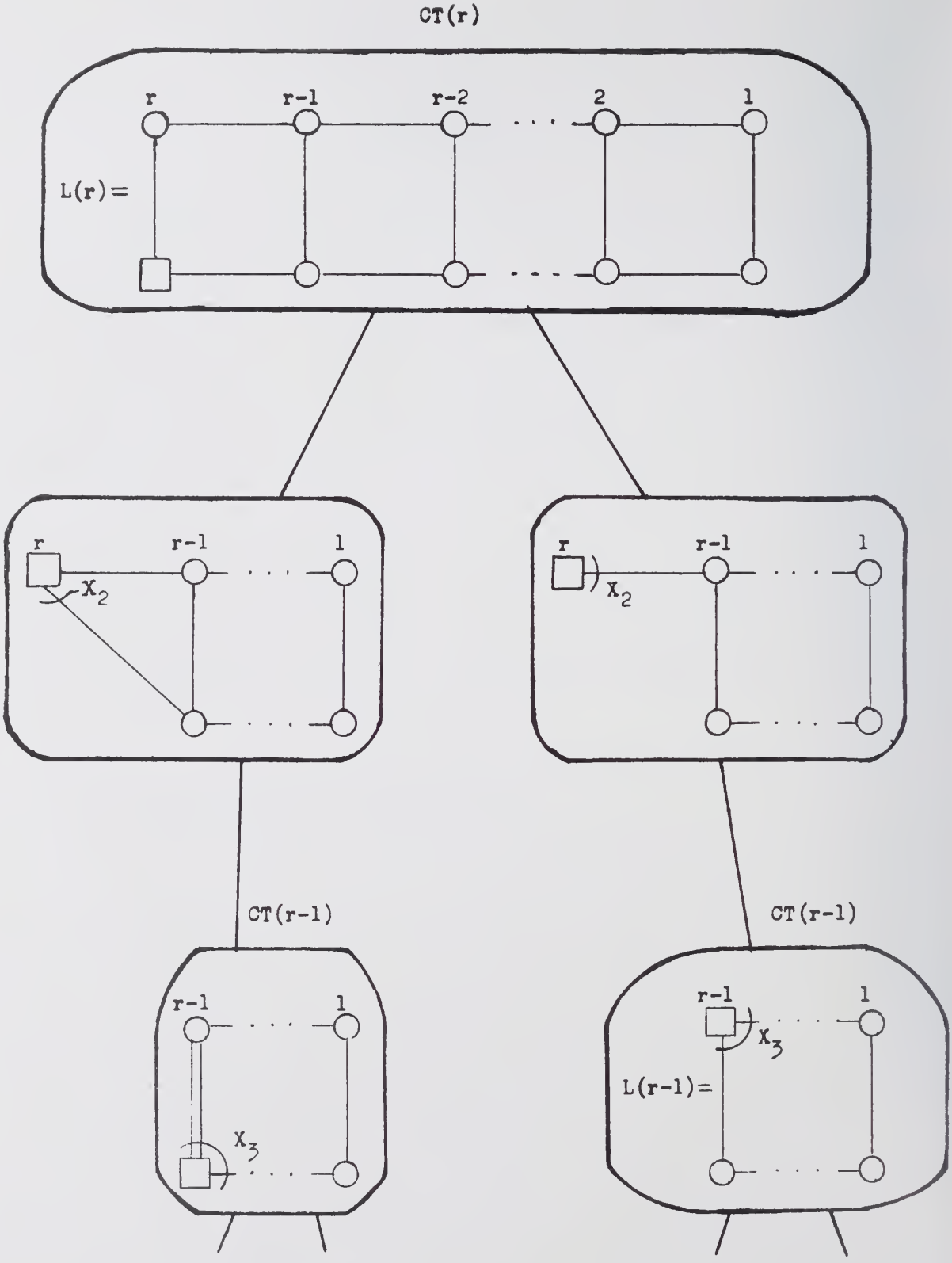

Figure 9: L(r) and CT(r, New)

node of $CT(r)$ immediately below this iteration of X_2, then $ST(r, w_3')$ is isomorphic to $CT(r)$.

From the above information, it is easy to derive the relationships $c_4(r) = 2 \cdot c_4(r-1)$, $c_2(r) = 2 \cdot c_2(r-1) + 3$, and $c_4(1) = c_2(1) = 1$. Applying the quotient operator and taking limits yields $q(c_i, r) = 2$. The claim is proved and $c_i(r, New) = O(2^r)$.

6.6. Complete Graphs

This section will measure the computation tree of algorithms with factoring [5.10, 5.11] applied to the complete graph on n nodes ($G = S(n)$). This test is significant as an "upper bound"; that is, for all graphs G on n nodes, $c_i(G, A) \leq c_i(S(n), A)$. Because of factoring, this inequality remains true even if G has parallel branches. For non-factoring algorithms, the statement would have to be modified to exclude parallel branches.

$CT(S(n), Fac)$ has $n-1$ iterations (branches) at the highest level because in $S(n)$, the starting node p_1 is adjacent to each of the $n-1$ other nodes. As these iterations are executed, Y_1 [the set of available branches in $S(n)$] will be diminished only by branches incident to p_1 [see figure 5]. Therefore, Y_1 will always contain a complete graph on the $n-1$ nodes excluding p_1. This completeness implies that for each node w at the second level of $CT(n, Fac)$, $ST(n, Fac, w)$ is isomorphic to $CT(n-1, Fac)$.

From this information, it is easy to derive the following relationships: $c_4(n, Fac) = (n-1)\, c_4(n-1, Fac)$; $c_2(n, Fac) = (n-1)\, c_2(n-1, Fac) + 1$; $c_2(1) = c_2(2) = c_4(1) = c_4(2) = 1$. Obviously, $c_4(n, Fac) = (n-1)!$, so $q(c_4, n) = n-1$. $Q(c_2, n) = [(n-1)c_2(n-1) + 1]/c_2(n-1) = n-1 + 1/c_2(n-1)$.

Since $\lim_{n\to\infty} \dfrac{n-1 + 1/c_2(n-1)}{n-1} = 1$, $q(c_2, n) = n-1$. Thus the quotient limits for c_2 and c_4 are equal. The only effect of the "+1" term in the recursive formula for c_2 is that $\lim_{n\to\infty} \dfrac{c_2(n)}{c_4(n)} = e$.

The complete graph analysis of the New algorithm is slightly more complicated. In box 2 [figure 6], if d_j = YES, then X_j is calculated as in the previous Factoring algorithm [figure 5]. The analysis of this (d_j = YES) case is similar to the analysis of the Factoring algorithm; if w_j is the corresponding node in CT(n, New), then ST(n, New, w_j) is isomorphic to CT(n+1-j, New).

Since d_1 = YES, X_1 will reach each of the n-1 nodes adjacent to p_1, the starting node. Let w_1 be the root of CT(n, New) and let w_2^i (i = 1, ..., n-1) be the nodes at level 2. For w_2^{n-1}, d_2 = YES, so ST(n, New, w_2^{n-1}) is isomorphic to CT(n-1, New). For $1 \le i \le n-2$, let $w_3^{i,k(i)}$ be the nodes at level 3 directly connected to w_2^i. For w_2^i ($1 \le i \le n-2$), d_2 = NO, but for $w_3^{i,k(i)}$, d_3 = YES: so ST(n, New, $w_3^{i,k(i)}$) is isomorphic to CT(n-2, New). To compute $\sum_{i=1}^{n-2} k(i)$, note that X_2^i (corresponding to w_2^i) is assigned [box 2, figure 6] the current (depleted) value of X_1. Thus there will be as many iterations of X_2^i [k(i)] as there are remaining iterations of X_1 [n-1-i]. Thus $\sum_{i=1}^{n-2} k(i) = \sum_{i=1}^{n-2} n-1-i = (n-2)(n-1)/2$.

From this information, it is easy to derive recursive formulas for $c_i(n, \text{New})$: $c_4(n) = c_4(n-1) + \frac{(n-2)(n-1)}{2} c_4(n-2)$; $c_2(n) = c_2(n-1) + \frac{(n-2)(n-1)}{2} c_2(n-2) + n-1$; $c_2(1) = c_2(2) = c_4(1) = c_4(2) = 1$; $c_2(3) = 3$.

Applying the quotient operator to these recursive equations yields $Q(c_4, n) = 1 + \frac{(n-2)(n-1)}{2 \cdot Q(c_4, n-1)}$. To show that $q(c_4, n) = (n-1)/\sqrt{2}$, let $m = \lim_{n\to\infty} Q(c_4, n) \cdot \frac{\sqrt{2}}{n-1}$. From the previous equation, $Q(c_4, n) \cdot \frac{\sqrt{2}}{n-1} = \frac{\sqrt{2}}{n-1} + \frac{n-2}{\sqrt{2}\, Q(c_4, n-1)}$. Taking the limit as $n\to\infty$, $m = 0 + \frac{1}{m}$; i.e., $m = 1$, so by definition, $q(c_4, n) = (n-1)/\sqrt{2}$. The proof that $q(c_2, n) = (n-1)/\sqrt{2}$ is

similar (an additional term goes to zero).

The quotient limit for the Circuit Check algorithm [6.4] can be derived from the fact that for $S(n)$, $b = n(n-1)/2$. Thus $c_4(S(n), CtC) = (\frac{2b}{n})^{n-1} = (n-1)^{n-1}$. $Q(c_4, n) = \frac{(n-1)^{n-1}}{(n-2)^{n-2}} = (n-1)(\frac{n-1}{n-2})^{n-2}$. Since $\lim_{n\to\infty} (\frac{n-1}{n-2})^{n-2} = e$, $q(c_4, n) = (n-1)e$.

The quotient limit for $t(n) = n^{n-2}$ is similarly derived: $Q(t, n) = \frac{n^{n-2}}{(n-1)^{n-3}} = (n-2+\frac{1}{n})(\frac{n}{n-1})^{n-1}$; $\lim_{n\to\infty} \frac{(n-2+\frac{1}{n})(\frac{n}{n-1})^{n-1}}{(n-2)e} = 1$; therefore, $q(t,n) = (n-2)e$.

Figure 10 plots $c_4(n, CtC)$, $t = c_4(n, CFE) = c_4(n, Con)$, $c_i(n, Fac)$, and $c_i(n, New)$ using the quotient operator $Q(f, n)$. The quotient limits derived above are the asymptotic limits of the plotted functions.

From this analysis, it is obvious that the cost per tree, c/t, goes to zero exponentially (e^{-n}) for the Factoring algorithm, and the cost ratio of the New algorithm to Factoring also goes to zero exponentially $(\sqrt{2}^{-n})$. Thus, the most efficient way to find trees one at a time is to use the New algorithm combined with a simple Cartesian product expansion algorithm [cost (New) < cost (simple expansion) < cost (other "one at a time" algorithms)].

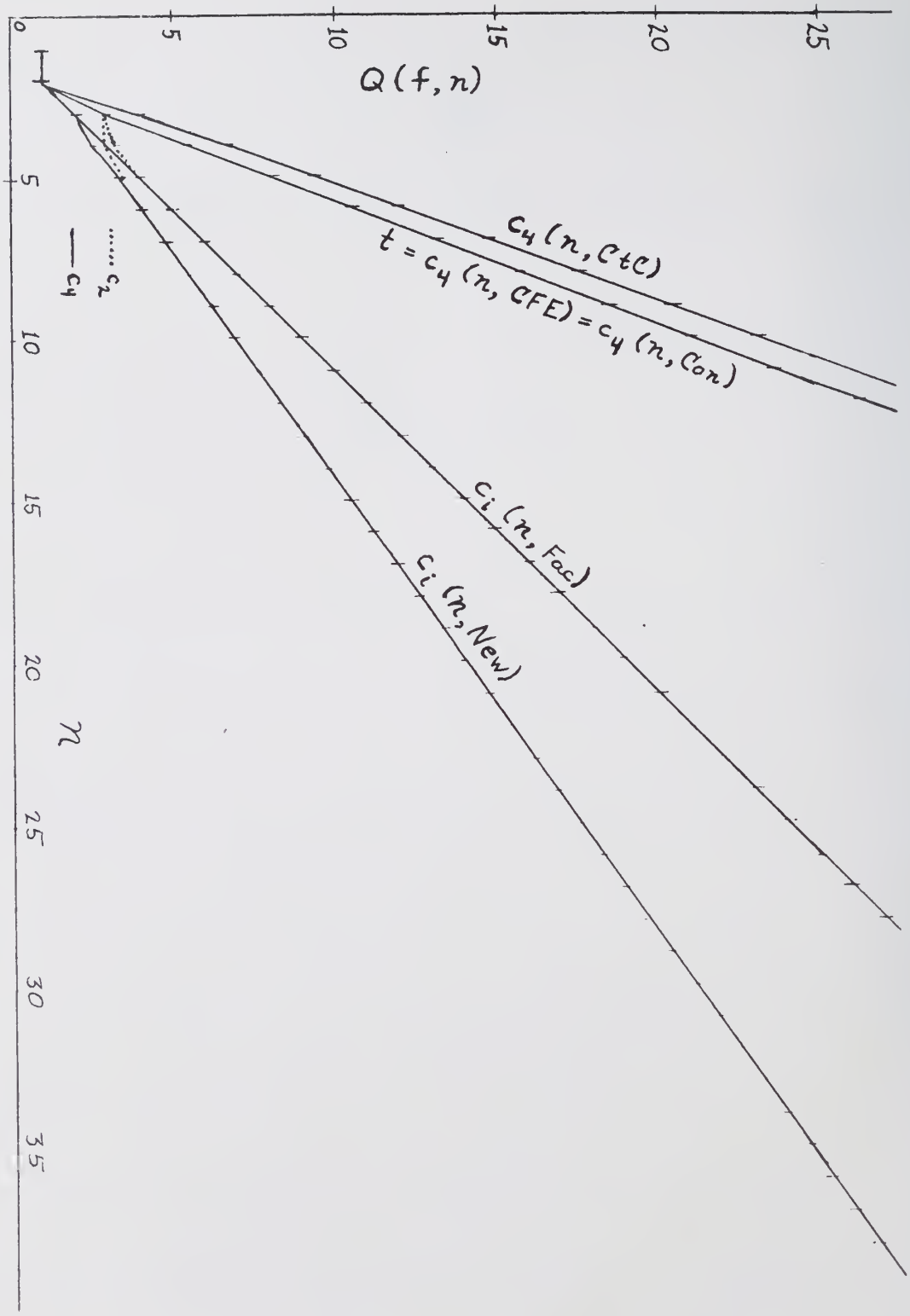

Figure 10: Complete Graph Analysis

7. CONCLUSIONS

One important contribution of this thesis is the efficiency analysis of all published algorithms for finding all trees of a graph. A very rough summary of this analysis is as follows:

Algorithm	Cost
"check for duplications"	t^2
"one tree at a time"	t
Factoring	te^{-n}
New	$t(e\sqrt{2})^{-n}$

Note that for the algorithms with factoring, the cost per tree goes to zero exponentially as n increases.

The techniques which were used to measure efficiency include the following: (1) the use of special classes of graphs on which the cost of an algorithm can be accurately measured (e.g., complete graphs); (2) the direct comparison (e.g., using computation trees) of competing algorithms in order to show differences in efficiency without the need to derive individual bounds; (3) the isolation of each idea of an algorithm (e.g., factoring) so that the efficient ideas can be available for the development and analysis of new algorithms; (4) the minimization of implementation details in empirical measurements (e.g., using GASP and counting statements rather than seconds); (5) the use of measures which reflect the nature of the class of algorithms (e.g., the quotient operator which linearizes the exponential nature of "recursive" algorithms).

The New algorithm is an important contribution of this thesis primarily because these techniques show that it is more efficient than any previous algorithm for finding all trees.

REFERENCES

These references are broken into three sections, and each section has its own aims and criteria for inclusion.

The first section aims at being an exhaustive list of papers which discuss various aspects of algorithms for finding all spanning trees of a graph. In addition, this section contains some references which discuss graph theoretical results of potential importance to such algorithms (e.g., the existence of a Hamiltonian circuit in the tree graph of a graph, or bounds on the number of trees in a graph). Several of these references discuss applications which use all spanning trees, mainly in the analysis of linear electrical networks.

The second section contains references to reports on general purpose graph-processing languages or software packages (programs which implement a specific graph algorithm are not included here).

The third section includes a few references to important papers on other graph algorithms, in particular those concerned with the following problems:

a) Find a minimum cost spanning tree of a graph.

b) Find a basis in the vector space of circuits of a graph (also known as a set of fundamental circuits).

c) Determine isomorphism of graphs.

d) Determine if a graph is planar.

e) Find shortest paths in a graph.

1. Spanning Trees

Amoia, V., and Cottafava, G. "On Central Trees," Proceedings of the 12th Midwest Symposium on Circuit Theory, paper XIV.1, 1969.

Ardon, M., and Malik, N. "A Recursive Algorithm for Generating Trees and Signed Complete Trees," Proceedings of the 12th Midwest Symposium on Circuit Theory, paper VII.2, 1969.

Bedrosian, S. "Application of Linear Graphs to Multilevel Maser Analysis," Journal of the Franklin Institute, Vol. 274, No. 4, pp. 278-283, October 1962.

Bellert, S. "Topological Analysis and Synthesis of Linear Systems," Journal of the Franklin Institute, Vol. 274, No. 6, pp. 425-443, December 1962.

Bercovici, M. "Formulas for the Number of Trees in a Graph," IEEE Transactions on Circuit Theory, Vol. CT-16, pp. 101-102, February 1969.

Berger, I. "The Enumeration of Trees Without Duplication," IEEE Transactions on Circuit Theory, Vol. CT-14, pp. 417-418, December 1967.

Berger, I., and Nathan, A. "The Algebra of Sets of Trees, K-Trees, and Other Configurations," IEEE Transactions on Circuit Theory, Vol. CT-15, pp. 221-226, September 1968.

Brownell, R. "Growing the Trees of a Graph," Proceedings of the IEEE, Vol. 56, pp. 1121-1123, June 1968.

Cayley, A. "A Theorem on Trees," Quarterly Journal of Mathematics, Vol. 23, pp. 376-378, 1889.

Chang, W., and Chan, S.G. "A Fast Tree-Finding Method," Proceedings of the 11th Midwest Symposium on Circuit Theory, pp. 457-462, 1968.

Char, J. "Generation of Trees, Two-Trees and Storage of Master Forests," IEEE Transactions on Circuit Theory, Vol. CT-15, pp. 228-238, September 1968.

Chen, W. "Generation of Trees and K-Trees," Proceedings of the Third Allerton Conference on Circuit and Systems Theory, pp. 889-899, 1965.

Chen, W. "On the Generation of Non-Singular Submatrices and Their Corresponding Subgraphs," Proceedings of the Fourth Allerton Conference on Circuit and Systems Theory, pp. 207-217, 1966a.

Chen, W. "On the Realization of Directed Trees and Directed 2-Trees," IEEE Transactions on Circuit Theory, Vol. CT-13, pp. 230-232, June 1966b.

Chen, W. "Hamilton Circuits in Directed-Tree Graphs," IEEE Transactions on Circuit Theory, Vol. CT-14, pp. 231-233, June 1967.

Chen, W. "Iterative Procedure for Generating Trees and Directed Trees," Electronic Letters, Vol. 4, No. 23, pp. 516-518, November 1968.

Chen, W. "Computer Generation of Trees and Co-Trees in a Cascade of Multiterminal Networks," IEEE Transactions on Circuit Theory, Vol. CT-16, pp. 518-526, November 1969a.

Chen, W. "Generation of Trees and Co-Trees of a Graph by Decomposition," Proceedings of the IEE (London), Vol. 116, No. 10, pp. 1639-1643, October 1969b.

Chen, W. "On the Generation of Trees Without Duplications," Proceedings of the IEEE, Vol. 57, pp. 1292-1293, July 1969c.

Chen, W., and Mark, S. "On the Algebraic Relationship of Trees, Co-Trees, Circuits, and Cutsets of a Graph," IEEE Transactions on Circuit Theory, Vol. CT-16, pp. 176-184, May 1969d.

Cummins, R., and Thomason, L. "An Efficient Tree-Listing Program," Unpublished, 1964.

Cummins, R. "Hamilton Circuits in Tree Graphs," IEEE Transactions on Circuit Theory, Vol. CT-13, pp. 82-90, March 1966.

Dawson, D. "Computational Aspects of the Topological Approach to Active Linear Network Analysis," Proceedings of Hawaii International Conference on System Sciences, pp. 113-115, 1968.

Dunn, W. Jr., and Chan, S.P. "Topological Formulation of Network Functions Without Generation of K-Trees," Proceedings of the Sixth Allerton Conference on Circuit and Systems Theory, pp. 822-831, 1968.

Fernandez, E. "Analisis de Redes Electricas con Computador Digital mediante Formulas Topologicas," Thesis, Departamento de Electricidad, Universidad de Chile, 1969a.

Fernandez, E. "An Evaluation of Tree Generation Methods," Proceedings of the 12th Midwest Symposium on Circuit Theory, paper VII.4, 1969b.

Feussner, W. "Uber Stromverzweigung in Netzformigen Leitern," Annalen der Physik, Vol. 9, pp. 1304-1329, 1902.

Feussner, W. "Zur Berechnung der Stromstarke in Netzformigen Leitern," Annalen der Physik, Vol. 15, pp. 385-394, 1904.

Fujisawa, T. "On a Problem of Network Topology," IRE Transactions on Circuit Theory, Vol. CT-6, pp. 261-266, September 1959.

Hakimi, S. "On Trees of a Graph and Their Generation," Journal of the Franklin Institute, Vol. 272, No. 5, pp. 347-359, November 1961.

Hakimi, S., and Green, G. "Generation and Realization of Trees and K-Trees," IEEE Transactions on Circuit Theory, Vol. CT-11, pp. 247-255, June 1964.

Hakimi, S., and Deo, N. "A Topological Approach to the Analysis of Linear Circuits," Proceedings of the Fourth Allerton Conference on Circuit and Systems Theory, pp. 197-206, 1966.

Hale, H. "A Logic for Identifying Trees of a Graph," AIEE Transactions on Power Apparatus and Systems, Vol. 80, pp. 195-197, June 1961.

Harary, F. "Graph Theory and Electrical Networks," IRE Transactions on Circuit Theory, Vol. CT-6, pp. 95-109, May 1959.

Hirayama, H., Watanabe, H., and Harada, K. "Digital Determination of Trees in Network Topology," Journal of the Institute of Electrical Communications Engineers of Japan, Vol. 46, No. 1, pp. 23-30, January 1963.

Hirayama, H., and Ohtsuki, T. "Topological Network Analysis by Digital Computer," Journal of the Institute of Electrical Communication Engineers of Japan, Vol. 48, No. 3, pp. 424-432, March 1965.

Hobbs, E., and MacWilliams, F. "Topological Network Analysis as a Computer Program," IRE Transactions on Circuit Theory, Vol. CT-6, pp. 135-136, March 1959.

Holt, A., and Fiedler, J. "Efficient Tree-Generation Method Suitable for Computer Programming," Electronic Letters, Vol. 4, No. 10, pp. 183-184, May 1968.

Jong, M., Lau, H., and Zobrist, G. "Tree Generation," Electronic Letters, Vol. 2, No. 8, pp. 318-319, August 1966.

Kamae, T. "The Existence of a Hamilton Circuit in a Tree Graph," IEEE Transactions on Circuit Theory, Vol. CT-14, pp. 279-283, September 1967.

Kim, W., Freiman, C., Younger, D., and Mayeda, W. "On Iterative Factorization in Network Analysis by Digital Computers," Eastern Joint Computer Conference, pp. 241-253, December 1960.

Kishi, G., and Kajitani, Y. "On Maximally Distinct Trees," Proceedings of the Fifth Annual Allerton Conference on Circuit and Systems Theory, pp. 635-643, 1967.

Kishi, G., and Kajitani, Y. "On Hamilton Circuits in Tree Graphs," IEEE Transactions on Circuit Theory, Vol. CT-15, pp. 42-50, March 1968.

Kishi, G., and Kajitani, Y. "Maximally Distant Trees and Principal Partition of a Linear Graph," IEEE Transactions on Circuit Theory, Vol. CT-16, pp. 323-330, August 1969.

Lee, S. "On Topological Formulae," Proceedings of the First Annual Allerton Conference on Circuit and Systems Theory, pp. 427-455, 1963.

MacWilliams, J. "Topological Network Analysis as a Computer Program," IRE Transactions on Circuit Theory, Vol. CT-5, pp. 228-229, September 1958.

Malik, N., and Lee, Y. "Finding Trees and Signed Tree-Pairs by the Compound Method," Proceedings of the 10th Midwest Symposium on Circuit Theory, paper VI-5, 1967.

Mason, S. "Topological Analysis of Linear Non-Reciprocal Networks," Proceedings of the IRE, Vol. 45, pp. 829-838, June 1957.

Maxwell, L., and Cline, J. "Topological Network Analysis by Algebraic Methods," Proceedings of the IEE (London), Vol. 113, No. 8, pp. 1344-1347, August 1966.

Mayeda, W. "Digital Determination of Topological Quantities and Network Functions," Interim Technical Report No. 6, Contract No. DA-11-022-ORD-1983, University of Illinois, Urbana, Illinois, January 1957.

Mayeda, W. "Reducing Computational Time in the Analysis of Networks by Digital Computers," IRE Transactions on Circuit Theory, Vol. CT-6, pp. 136-137, March 1959.

Mayeda, W., and Seshu, S. "Generation of Trees Without Duplications," IEEE Transactions on Circuit Theory, Vol. CT-12, pp. 181-185, June 1965.

Mayeda, W. "Generation of Trees and Complete Trees," CSL Report R-284, University of Illinois, Urbana, Illinois, April 1966.

Mayeda, W., Hakimi, S., Chen, W. and Deo, N. "Generation of Complete Trees," IEEE Transactions on Circuit Theory, Vol. CT-15, pp. 101-105, June 1968.

McIlroy, M. "Generator of Spanning Trees," Communications of the ACM, Vol. 12, No. 9, p. 511, September 1969.

Minty, G. "A Simple Algorithm for Listing All the Trees of a Graph," IEEE Transactions on Circuit Theory, Vol. CT-12, p. 120, March 1965.

Mullin, R., and Stanton, R. "A Combinatorial Property of Spanning Forests in Connected Graphs," Journal of Combinatorial Theory, Vol. 3, pp. 236-243, 1967.

Myers, B., and Auth, L. Jr. "The Number and Listing of All Trees in an Arbitrary Graph," Journal of Combinatorial Theory, Vol. 3, pp. 236-243, 1967.

Myers, B., and Auth, L. Jr. "The Number and Listing of All Trees in an Arbitrary Graph," Proceedings of the Third Allerton Conference on Circuit and Systems Theory, pp. 906-912, 1965.

Myers, B. "Efficient Generation of Tree-Admittance Products in a Cascade of 2-Port Networks," Proceedings of the IEE (London), Vol. 114, No. 11, pp. 1641-1646, November 1967.

Nakagawa, N. "On Evaluation of the Graph Trees and the Driving Point Admittance," IRE Transactions on Circuit Theory, Vol. CT-5, pp. 122-127, June 1958.

O'Neil, P., and Slepian, P. "An Application of Feussner's Method to Tree Counting," IEEE Transactions on Circuit Theory, Vol. CT-13, pp. 336-339, September 1966a.

O'Neil, P., and Slepian, P. "The Number of Trees in a Network," IEEE Transactions on Circuit Theory, Vol. CT-13, pp. 271-281, September 1966b.

Paul, A. Jr. "Generation of Directed Trees and 2-Trees Without Duplications," IEEE Transactions on Circuit Theory, Vol. CT-14, pp. 354-356, September 1967.

Percival, W. "The Solution of Passive Electrical Networks by Means of Mathematical Trees," Proceedings of the IEE (London), pt. 3, Vol. 100, pp. 143-150, May 1953.

Piekarski, M. "Listing of All Possible Trees of a Linear Graph," IEEE Transactions on Circuit Theory, Vol. CT-12, pp. 124-125, March 1965.

Riordan, J. "The Enumeration of Trees by Height and Diameter," IBM Journal of Research and Development, Vol. 4, No. 5, pp. 473-478, November 1960.

Roe, P. "On a Tree Expansion Theorem," IRE Transactions on Circuit Theory, Vol. CT-8, pp. 496-500, December 1961.

Scoins, H. "Placing Trees in Lexicographic Order," Machine Intelligence 3, pp. 43-60, 1968.

Shank, H. "A Note on Hamilton Circuits in Tree Graphs," IEEE Transactions on Circuit Theory, Vol. CT-15, p. 86, March 1968.

Shelman, C., Maenpaa, J., and Stahl, W. "Complete Tree Generation - Some Practical Experience," IEEE Transactions on Circuit Theory, Vol. CT-16, pp. 548,550, November 1969.

Trent, H. "A Note on Enumeration and Listing of All Possible Trees in a Connected Linear Graph," Proceedings of the National Academy of Science, Vol. 40, pp. 1004-1007, October 1954.

Watanabe, H. "Computational Method for Network Topology," IRE Transactions on Circuit Theory, Vol. CT-7, pp. 296-302, September 1960.

Watanabe, H. "A Method of Tree Expansion in Network Topology," IEEE Transactions on Circuit Theory, Vol. CT-8, pp. 4-10, March 1961.

Weinberg, L. "Kirchoff's Third and Fourth Laws," IRE Transactions on Circuit Theory, Vol. CT-5, pp. 8-30, March 1958.

Wing, O. "Enumeration of Trees," IEEE Transactions on Circuit Theory, Vol. CT-10, pp. 127-128, March 1963.

Zobrist, G., and Lago, G. "Digital Computer Analysis of Passive Networks Using Topological Formula," Proceedings of the Second Annual Allerton Conference on Circuit and Systems Theory, pp. 573-595, 1964.

2. General Purpose Graph Software

Christensen, C. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language," Interactive Systems for Experimental Applied Mathematics, pp. 423-435, 1968.

Friedman, D.P., Dickson, D.C., Fraser, J.J., and Pratt, T.W. "GRASPE 1.5- A Graph Processor and its Application," University of Houston Report RS 1-69, Houston, Texas, August 1969.

Hart, R. "HINT: A Graph Processing Language," Research Report, Computer Institute for Social Science Research, Michigan State University, East Lansing, Michigan, February 1969.

Read. R.C., King, C., Cadogan, C.C., and Morris, P. "The Application of Digital Computer Techniques to the Study of Graph Theoretical and Related Combinatorial Problems," Scientific Report, Computing Centre, University of West Indies, Mona, Kingston 7, Jamaica.

Wolfberg, M.S. "An Interactive Graph Theory System," Moore School Report No. 69-25, University of Pennsylvania, Philadelphia, Pennsylvania, June 1969.

3. Selected References to Other Graph Algorithms

Chase, S. "How to Win Shannon Switching Games: A Case Study in Automatic Graph Processing," Communications of the ACM, (to appear).

Corneil, D., and Gotlieb, C. "An Efficient Algorithm for Graph Isomorphism," Journal of the ACM, Vol. 17, No. 1, pp. 51-64, January 1970.

Dijkstra, E. "A Note on Two Problems in Connection with Graphs," Numerische Mathematik, Vol. 1, No. 5, pp. 269-271, October 1959.

Gotlieb, C., and Corneil, D. "Algorithms for Finding a Fundamental Set of Cycles for an Undirected Linear Graph," Communications of the ACM, Vol. 10, No. 12, pp. 780-783, December 1967.

Paton, K. "An Algorithm for Finding a Fundamental Set of Cycles of a Graph," Communications of the ACM, Vol. 12, No. 9, pp. 514-518, September 1969.

Shirey, R. "Implementation and Analysis of Efficient Graph Planarity Testing Algorithms," Ph.D. Thesis, University of Wisconsin, Madison, Wisconsin, 1969.

Unger, S.H. "GIT - A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism," Communications of the ACM, Vol. 7, No. 1, pp. 26-34, January 1964.

Witzgall, C. "On Labelling Algorithms for Determining Shortest Pahts in Networks," NBS Report 9840, 1968.

APPENDICES

1. GASP MANUAL

1.1. Purposes of GASP

The main purpose the the Graph Algorithm Software Package is to allow programmers of graph algorithms to code programs in a natural and machine independent way. Because operations are expressed in a language of graph and set terms, the programs will be easy to follow and estimates of the amount of computation will be easier to compute. Comparisons among different algorithms for the same problem will be much easier using GASP becuase it is easy to generate programs from their description in conventional graph-theoretic terms. Moreover, some tallies on the amount of computation are provided by GASP.

The logical structure of GASP makes it possible to change representations with relatively little change in programs. A few low-level routines would have to be rewritten, but the higher level programs would not.

1.2. Basic Concepts and Terminology

1.2.1. Data Types

An integer has the usual definition. A character string is a fixed-length string of characters. A truth value is a variable which can take on one of the two values: yes or no. A name references an object (see below). An object is a conglomeration consisting of one integer, one character string, three names, and (most important) one set. A set can exist only as part of an object. There are two types of objects: restricted and unrestricted. Restricted objects cannot belong to sets.

1.2.2. Definitions and Assignments

GASP objects are available to the programmer only through one level of

indirect addressing. The programmer deals with names which refer to objects which have values. This relationship (shown in figure 11) is very important for the understanding of GASP.

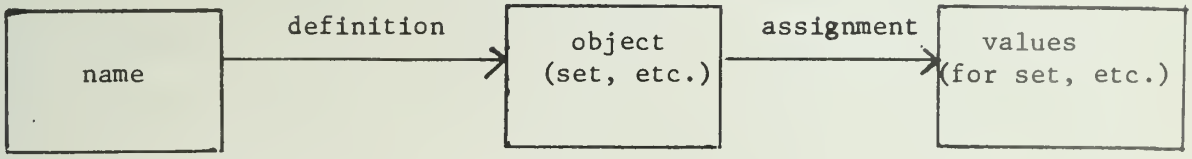

Figure 11

In order to distinguish one level from the other, we will use two sets of words. A name is defined if it references an existing object, otherwise it is undefined. A name may change its definition; that is, it can be made to refer to a different object. The number of names referring to an object may vary from zero to any reasonable positive integer.

When the contents of a set are changed, we say the set is assigned a new value. Also, the object involved is assigned a new value.

The main advantage of this indirect addressing scheme is that not all objects need to be accessed by permanent names. E.g., all objects in a set S may be made accessible by means of the statement "FOR (ALL, X, S)", even if no object in S has previously been given a name. In this case, we regard the bound variable X as a name whose definition ranges over all the objects in the set S.

1.2.3. Graphs

A graph is represented as an object whose set contains the branches and nodes belonging to the graph. Each node and each branch is an unrestricted object. In this first implementation, the set associated with a branch is the

set of two incident nodes; the set associated with a node is the set of all incident branches and adjacent nodes.

1.2.4. System Objects

GASP reserves a few restricted objects for special use. NULLSET is a read-only object whose set is always empty. AC is an object whose set holds intermediate values of set operations. USED is the object whose set contains all currently active unrestricted objects. NODES and BRANCHES are the objects whose sets contain all nodes and branches (respectively) belonging to the union of all graphs.

1.2.5. GASP Statement Forms

GASP is an extension of PL/1 (through the use of the PL/1 Preprocessor). Thus any PL/1 statement could be considered a GASP statement. The 'pure' GASP statements fall into three categories: PL/1 statements which declare or assign values to GASP data types; GASP Procedure calls which constitute a complete statement starting with CALL (unless the procedure name begins with $) and ending with a semi-colon; Type-functions where type is one of the following: name, integer, truth value. A type-function call can be inserted almost anywhere a 'type' variable is allowed.

1.3. The GASP Statements for Sets

1.3.1. Notation

In the instruction set that follows, the actual code that must appear (as spelled) is capitalized while arbitrary names used as arguments are not. GASP statements will frequently be set off by quotation marks which are obviously not part of the code.

1.3.2. Declarations

GASP variables are declared just as regular PL/1 variables are declared. Conversion from terms in section 2 to actual program words is shown below.

Formal Description	Computer Code
name (and unrestricted object)	$NAME
integer	$INTEGER
character string	$CHAR
truth value	$BIT
restricted object	$MAXSET

NOTE: Declaring a name (or unrestricted object) does not define it. However, 'DCL x $MAXSET;' will create a restricted object and that object will be the definition of x. $MAXSET is the only declaration which cannot be factored; 'DCL (S1, S2) $MAXSET;' would result in both names S1 and S2 referring to the same object.

1.3.3. Definitions

A name, x, can be defined in two other ways besides 'DCL x $MAXSET;' [previous paragraph].

'$ALLOC (x) ;' creates (storage for) a new unrestricted object which will become the definition of the name x (previously declared $NAME). Also the character string part of this object will be assigned the value 'x'.

'x = name-expression ;' will define (or redefine) the name x to refer to the object named by name-expression (name-expression can be either a previously defined name or an arbitrary expression which computes a name value).

1.3.4. Freeing of Storage

The storage taken up by an object can be freed as follows:

'$KILL (x) ;' will free the unrestricted object named x and the name x will become undefined.

'CALL POP (x) ;' will free the restricted object named x and leave x undefined.

1.3.5. Operations on Sets

1.3.5.1. Notation

Nearly all arguments will be defined names. In the following examples, those names beginning with 's' are to be considered as sets (of any object) and those beginning with 'e' should be considered as elements of sets ('e' names must refer to unrestricted objects). As is the case with most GASP operations, no names are changed by the instructions in this section.

1.3.5.2. Truth Value Functions

'$IS-IN(e, s)' answers "does e belong to s?".

'$EQUALS(s1, s2)' answers "does set s1 = set s2?".

'$EMPTY_(s)' is equivalent to '$EQUALS(s, NULLSET)'.

1.3.5.3. Procedure Calls

'$STORES (sname, s_expression) ;' will assign to the set named sname the value of the set named s_expression (which remains unchanged).

'$CLEAR(s);' is equivalent to '$STORES (s, NULLSET) ;'.

'$CHANGES (s, elem, op)', where op = ADD or DELETE, will add (delete) elem to (from) the set s. If this does not change the truth value of the expression "elem $\in$ s", then it is a harmless waste of time.

'$CSES (s, e) ;' (Clear and Store Element in Set) assigns to s the value {e}.

1.3.5.4. Integer Functions

'CARD(s)' returns the integer number of elements belonging to s.

1.3.5.5. Name Functions (Choice)

The functions in this section pick elements out of sets with varying

side effects.

'ELEM_OF(s)' will return the name of an object belonging to the set s. This statement should not be used unless it is known that s is not empty. The set s is unchanged by this instruction.

'CAN_PIC (e, s)' is a truth value function which will answer "CAN one PICk an element from s?". If set s is not empty, e will name an object belonging to s, which will then be deleted from s. If s is empty, e will be undefined.

'ITH_EL (s, i)' will return (withoug deleting) the i-th element of the set s, where i is an integer. Since this depends on the arbitrary (but fixed) ordering of elements in the implementation of the set, it has little use.

'RANDEL(s)' will return a randomly chosen element from the set s, without deleting it.

1.3.5.6. Name Functions (Intermediate Results)

The name functions in this section all perform some operation on the input sets and store the result in the AC. The name returned is always AC. The input sets remain unchanged (unless one of them is AC).

'UNION (s1, s2)' takes the union of sets s1 and s2.

'INTER (s1, s2)' takes the intersection of sets s1 and s2.

'COMPL (s)' takes the complement of the set s with respect to the universal set of unrestricted objects (useless by itself).

'DIFF (s1, s2)' contains those objects belonging to s1 but not to s2.

'SYMDIF (s1, s2)' contains those objects belonging to exactly one of the sets s1 and s2 (exclusive or).

1.3.6. Saving Object Values

'CALL PUSH (x) ;' saves the value of the object named x.

'CALL POP (x) ;' restores the saved value of the object named x. For example, consider the following code:

```
. . .
1
CALL PUSH (x);
2
. . .
3
CALL POP (x);
4
. . .
```

The values (of all the parts) of the object named x will be the same at points 1, 2, and 4 regardless of the values at point 3. The definition of x remains unchanged throughout.

As the words 'push' and 'pop' imply, any number of copies of an object may be saved in this way, and restored in the usual 'last in - first out' order. Implementation restrictions will limit the number of saved objects at any point during execution.

1.3.7. I/O

GASP does not aid the user in the input of sets.

'CALL PELEMSK (s);' (Print ELEMent and SKip to next line) will print the character string of the object s and the character string of all objects belonging to the set of s.

EXAMPLE:

```
DCL (S1, S2, E1, E2, E3, E4) $NAME;
$ALLOC (S1) ; $ALLOC (E1); $ALLOC (E2);
E3 = E1; E4 = E2; S2 = S1;
$CSES (S2, E3); $CHANGES (S2, E4, ADD);
```

CALL PELEMSK (S2); END;

would generate the output line

S1 = (E1, E2)

and skip to the next line.

'$PUT (var-name);' is like 'PUT DATA (var-name);' but with no restrictions on var-name.

'CALL ABDUMP;' dumps the entire data base (of objects).

Regular PL/1 I/O is also available.

1.3.8. Expanding Operations

'EXPAND2 (subr, set)' is a name function which returns AC. Subr may be any name function (e.g., UNION, INTER, SYMDIF) which takes two names as arguments and performs a binary (usually associative and communtative) operation on their sets, returning the name of the set which holds the result. For example, if s = {e1, e2, e3}, then EXPAND2 (UNION,s) is equivalent to UNION(e1, UNION (e2, e3)). If s is empty, then EXPAND2 (subr, s) is empty, and if s = {e1} then EXPAND2 (subr, s) = (the value of the set) e1.

'CALL EXPAND1 (subr, set);' is a procedure call which can be used with any procedure subr which takes one name as input. EXPAND1 will call this routine with set as the argument, and then will call it with each object belonging to set as the argument. Useful choices for subr include PELEMSK, PUSH, and POP.

1.3.9. Loop Control

'$FOR (q, x, s);'

code

'$END;'

allows code to be executed iteratively with each iteration having a different definition of the name x chosen from the set s. The quantifier, q, may

be any number, including ANY (equivalent to 1) and ALL (equivalent to cardinalit of set s). Code will be executed minimum (q, ALL) times. 's' may be any name or name function. Once the $FOR statement is executed, changing s will not affect or be affected by the iterations. The $FOR - $END pair is a PL/1 block, and the bound variable x is automatically declared within this block (it need not be declared before).

The normal exit from a $FOR - $END section is to the next statement after $END. Any other jump outside must be expressed as 'GO_TO label ;'.

'$4ALLPAIRS (bv1, bv2, s)'

code

'$4APEND;'

is similar to the $FOR statement except that code is executed for all possible unordered choices of bound variables bv1 and bv2 subject to bv1, bv2 $\in$ s and bv1 $\neq$ bv2. Abnormal exits must be through the statement 'GO_2 label;'.

1.4. The GASP Statements for Graphs

Nodes will be denoted by n, n1, n2; branches by b, graphs by g.

1.4.1. Truth Value Functions

'INCIDENT (n, b)' answers "is n incident to b?".

'ADJACENT (n, n2)' answers "is n adjacent to n2?".

1.4.2. Simple Information Extraction

To get the set of adjacent nodes or incident branches of a given n or the set of incident nodes of a given b, use the following name functions (all return AC):

'SET_OF_INCIDENT (NODES, n)',

'SET_OF_INCIDENT (BRANCHES, n)', or

'SET_OF_INCIDENT (NODES, b)'.

To get the set of nodes in g, use the name function (returns AC) '$NOF(g)'. Similarly, the branches of g are obtained by '$BOF(g)'.

'CALL GET_BAN (b, n1, n2, g) ;' defines n1 and n2 to be the endpoints of b in g.

1.4.3. Advanced Graph Operations

'NBOUND (nodeset, g)' is a name function which returns (AC) the set of all nodes of g which do not belong to nodeset but are adjacent to at least one node belonging to nodeset. 'BBOUND (nodeset, g)' is a name function which returns (AC) the set of all branches of g which have exactly one endpoint belonging to nodeset.

'CALL INTBANS (s, g) ;' is a procedure call which returns with s reassigned the value of the subset of branches of g which have both endpoints belonging to s (a set of nodes at input time).

'DIST (n1, n2, g)' is an integer function which returns the distance from n1 to n2 in g.

'CALL COLAPS (b, g) ;' is a procedure call which changes g by merging the endpoints of b into a single node and removing any branches connecting those endpoints (such as b).

'CALL DELBAN (b, g) ;' is a procedure call which deletes all trace of b from g.

'CALL DELNOD (n, g) ;' is a procedure call which deletes n and all of its incident branches from g.

1.4.4. Graph I/O

'CALL READGR (g) ;' is the procedure call to input g. The input format is a sequence of paths of node numbers (from 1 to the number of nodes). A new path is begun by a minus sign in front of the starting node [only Euler graphs can be given by a single path]. The entire sequence is terminated by a zero. READGR also will output g (see below).

EXAMPLE: Given the output sequence

1, 2, 4, -2, 3, 4, 1, 0,

READGR would create the graph shown in figure 12.

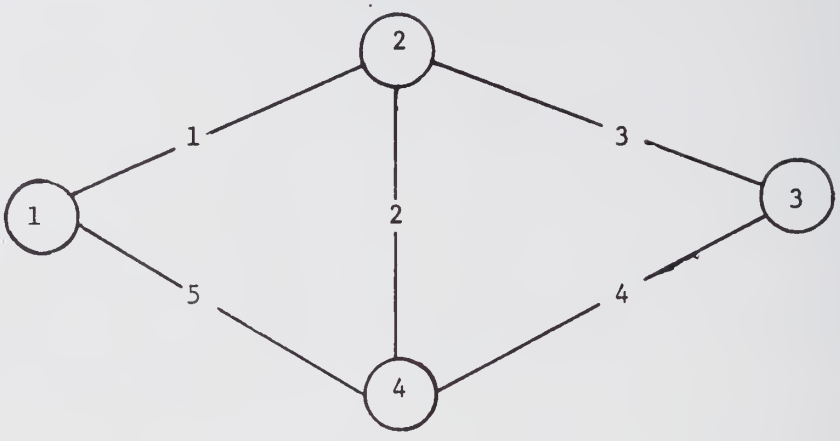

Figure 12

'CALL DEF_BAN (b, n1, n2, g) ;' is a procedure call which will create a branch b connecting n1 and n2 in g.

'$PUTGRAPH (g) ;' is the procedure call which outputs g as a set of nodes and branches. It is equivalent to 'CALL EXPAND1 (PELEMSK, g) ;'.

1.5. Measuring GASP Programs

A count of the number of executions of each block of a GASP program is accomplished with the following statements [even though they are complete statements, they need not be followed by a semicolon].

'$DCLSTAT(k)' declares k integers to be used for counting.

'$STAT' is placed in each logical block to be counted.

'$CLEARSTAT' initializes the counts to zero.

'$OUTSTAT' prints out the k integers, in the order that the '$STAT''s appeared (compilation-wise, not execution-wise).

1.6. Implementation Details

1.6.1. Data Structure

A Universal SET (USET) contains all GASP objects. USET is a PL/1 struc-

ture subdivided into $TSIZE objects (level name is ELEMENT) of which $SIZE are unrestricted. The current systems has $SIZE=64 and $TSIZE=127, but these can be changed easily. The set part (SSET) of an object is a bit string of length $SIZE (this is the only reason for restricted objects: an increase in the number of restricted objects increases the memory requirement only linearly; an increase in unrestricted objects increases memory requirements quadratically). The other parts of an object are CHARP (CHARacter string Part), INTP (INTeger Part), REFP (REFerence Part), RP_2 and RP_3 (Reference Part 2 and 3). PL/1 declarations for the various data types [1.2.1.] are as follows: $CHAR = CHAR (8), $INTEGER = BIN FIXED (15), $NAME = BIN FIXED, and $BIT = BIT (1).

1.6.2. How GASP Works

GASP procedures which require only a line or so of code are translated by the PL/1 preprocessor. The identifiers which are translated by the GASP macros usually start with a '$'.

Longer GASP procedures are incorporated into the programs as separate PL/1 procedures. The user has a choice of two methods which include these procedures in his program. The more efficient way is to include them as pre-compiled external procedures. The more flexible way is to have their source code inserted into the main program: this allows the user to set the limits $SIZE and $TSIZE to fit his needs.

1.6.3. Cost Parameters

Since the PL/1 preprocessor and PL/1 compiler are used, compilation time is usually large, run time is usually reasonable. For example, a typical program took 20 seconds to compile, 6 seconds to execute.

The core requirement for the basic GASP programs and data is around 120k bytes, a typical program might require a total of 150k bytes.

GASP macro definitions require 206 lines of PL/1 code, the source code of GASP procedures is around 240 lines.

1.6.4. Implementation Defects

When coding a binary set operation (e.g., 'UNION (S1, S2)'), one must make sure that at least one of the arguments is not AC.

'GO_TO' and 'GO_2' are precompiled into more than one PL/1 statement and therefore should not appear immediately after a 'THEN'.

2. THE NEW ALGORITHM PROGRAMMED IN GASP

```
NEW: PROC(G,N);  DCL G $NAME, N $INTEGER;
  DCL ( SET_PJ, XJ_NODES, AJ, XJ, P(N) ) $NAME,
  ( IS_DISCON, D(N) ) $BIT,   J        $INTEGER;
  %DCL ( KJ, ZJ, SAVED, $CARD ) CHAR;  /* USE PARTS OF OBJECTS */
  % KJ = 'INTP(XJ)' ;        % ZJ = 'REFP(AJ)' ; % $CARD = 'INTP' ;
  % SAVED = 'RP_2' ; /* RP_2 POINTS TO THE SAVED VALUE OF OBJECTS */
  IF N < 3 THEN RETURN ;                    $ALLOC(TEMP);
  $ALLOC(XJ); $ALLOC(SET_PJ); $ALLOC(XJ_NODES);$ALLOC(AJ);$ALLOC(YJ);
BOX1:  J = 1 ;
  P(1) = NODES#(1);   $CSES ( SET_PJ, P(1) ) ;
  $STORES ( XJ, SET_OF_INCIDENT ( BRANCHES, P(1))) ;
  $STORES ( YJ, $BOF ( G ) ) ;
  D(1) = YES ;
  IS_DISCON = NO ;
BOX2 :  $STAT   /* COUNT C2(G, NEW)  */
  IF D(J) THEN DO;  $STAT
  $ALLOC(ZJ);  $STORES( ZJ, XJ ) ;
  $STORES(XJ_NODES, DIFF( EXPAND2(UNION,XJ), SET_PJ ) ) ;
       /*    NODE BOUNDARY ( P1, P2, . . . , PJ )  */
  $CARD ( XJ_NODES ) = CARD ( XJ_NODES ) ;  END;
  KJ = 0 ;
BOX3 :  IF ¬CAN_PIC(P(J+1),XJ_NODES) THEN SIGNAL ERROR;
  $CARD ( XJ_NODES ) = $CARD ( XJ_NODES ) - 1  ;
  $STORES ( AJ, INTER ( XJ, P(J+1))) ;
  $STORES ( XJ, DIFF  ( XJ, AJ ) ) ;
```

```
   $STORES ( YJ, DIFF  ( YJ, AJ ) ) ;
   CALL PUSH (XJ) ;  /* X(J) <- X(J-1)  */
   $STORES ( XJ, UNION ( XJ, INTER(YJ,P(J+1)))) ;
    /* I.E.,X(J+1)<-  BOUND (P1, P2, . . . , P(J+1)) */
XJ_EMPTY:  IF $EMPTY_( XJ ) THEN DO;  $STAT
   CALL POP ( XJ ) ;  GO TO DISCON ;  END ;
   $CHANGES ( SET_PJ,P(J+1),ADD ) ;
   CALL PUSH ( YJ ) ;  /* Y(J+1) <- Y(J)  */
   CALL PUSH ( XJ_NODES ) ;
   IF ¬ D(J) THEN DO ;
   $STORES ( TEMP, DIFF ( AJ, ZJ ) ) ;
   IF ¬ $EMPTY_( TEMP ) THEN DO;  $STAT
     /* NOW PAY THE PRICE FOR INCORRECT XJ  */
   $STORES ( AJ, INTER ( AJ, ZJ ) ) ;
   $STORES ( SAVED ( YJ ), UNION(YJ, TEMP) ) ;
   $STORES ( SAVED ( XJ ), UNION(SAVED(XJ), TEMP) ) ;  END;
   D(J+1) = YES ;
   GO TO BUMP ;  END;
    /* ELSE IF D(J) THEN  */
   IF $CARD ( XJ_NODES ) > 0 THEN D ( J + 1 ) = NO ;
   ELSE DO; $STAT   D(J+1) = YES ;
   $STORES ( AJ, ZJ ) ;  $KILL(ZJ) ;  END;
BUMP:  KJ = KJ + 1 ;
   CALL PUSH(AJ) ;
   J = J + 1 ;
J_EQ_N :  IF J < N-1 THEN GO TO BOX2 ;
BOX4 :  $STAT  /* C4(G, NEW) */
```

```
  IF D(J) THEN $STORES ( AJ, YJ) ;
  ELSE  $STORES ( AJ, INTER(ZJ, YJ) ) ;
  /* 'OUTPUT A1 X A2 X . . . X A(N-1)' COMES HERE */
BOX5 :  J = J - 1 ;
  CALL POP(AJ); CALL POP(YJ); CALL POP(XJ); CALL POP(XJ_NODES);
  $CHANGES ( SET_PJ, P(J+1), DELETE ) ;
  IF IS_DISCON THEN GO TO DISCON ;
XJ_EMPTY_:  IF $CARD (XJ_NODES) > 0 THEN GO TO BOX3 ;
J_EQ_1 :  IF J = 1 THEN GO TO RETURN_ ;
  GO TO BOX5;
DISCON : $STAT
  IF D(J) THEN DO;  $KILL(ZJ);  $STAT  END;
  IS_DISCON = ( KJ = 1 ) ;
  GO TO J_EQ_1 ;
RETURN_ : $KILL(YJ); $KILL(AJ); $KILL(XJ); $KILL(XJ_NODES);
  $KILL(TEMP);  $KILL(SET_PJ);
  RETURN; END NEW;
```

VITA

The author, Stephen Martin Chase, was born in Urbana, Illinois, on September 21, 1943. He received his Bachelor of Science degree in Mathematics in June 1965, and his Master of Science degree in Mathematics in June 1967 from the University of Illinois. From June 1965 to June 1970, he was a research assistant in the Department of Computer Science of the University of Illinois at Urbana-Champaign. In June 1970, he joined the research staff of the Thomas J. Watson Research Center in Yorktown Heights, New York.